University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Pitt SCI

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 4: this Friday @ 11:59 pm
  - Lab 3: Tuesday 2/14 @ 11:59 pm
  - Assignment 1: Friday 2/17 @ 11:59 pm
- Please make your Piazza posts public as much as possible

Previous lecture

- Red-Black BST (self-balancing BST)
  - add
  - delete
  - runtime of operations
- Turning recursive tree traversals to iterative

This Lecture

- Binary Search Tree uses comparisons between keys to guide the searching
- What if we use the digital representation of keys for searching instead?
  - Keys are represented as a sequence of digits (e.g., bits) or alphabetic characters
- Digital Searching Problem

Digital Searching Problem

- Input:
  - a (large) dynamic set of data items in the form of
    - *n* (key, value) pairs; key is a string from an alphabet of size *R*
    - Each key has *b* bits or *w* characters (the chars are from the alphabet)
    - What is the relationship between *b* and *w*?
  - a *target key (k)*
- Output:
  - The corresponding value to *k* if target key found
  - Key not found otherwise

Digital Search Trees (DSTs)

Instead of looking at less than/greater than, lets go left or right based on the bits of the key

So, we again have 4 options:

- current node is null, k not found
- k is equal to the current node's key, k is found, return corresponding value
- current bit of k is 0, continue to left child
- current bit of k is 1, continue to right child

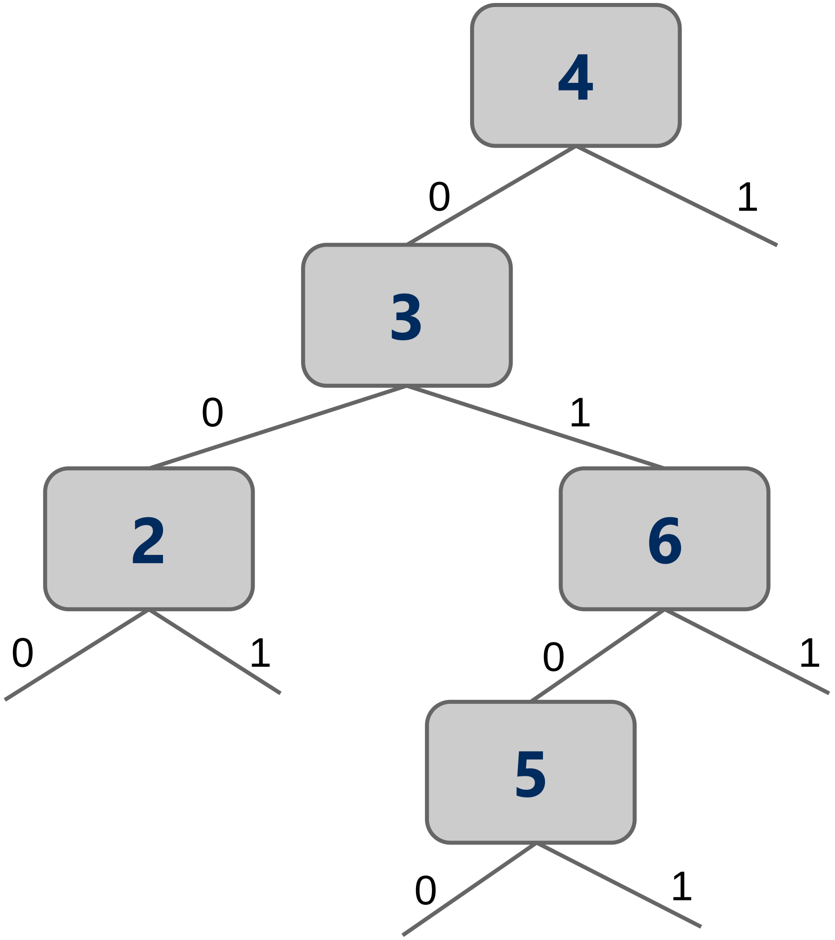
DST example: Insert and Search

Insert:

4	0100
3	0011
2	0010
6	0110
5	0101

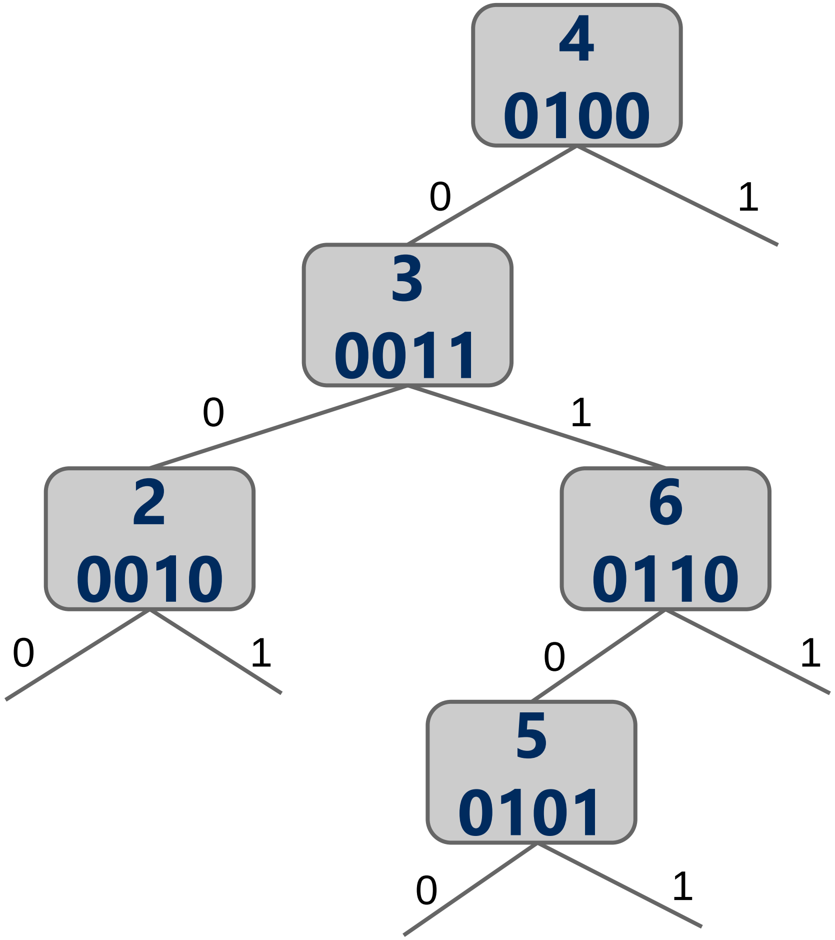
Search:

3	0011
7	0111



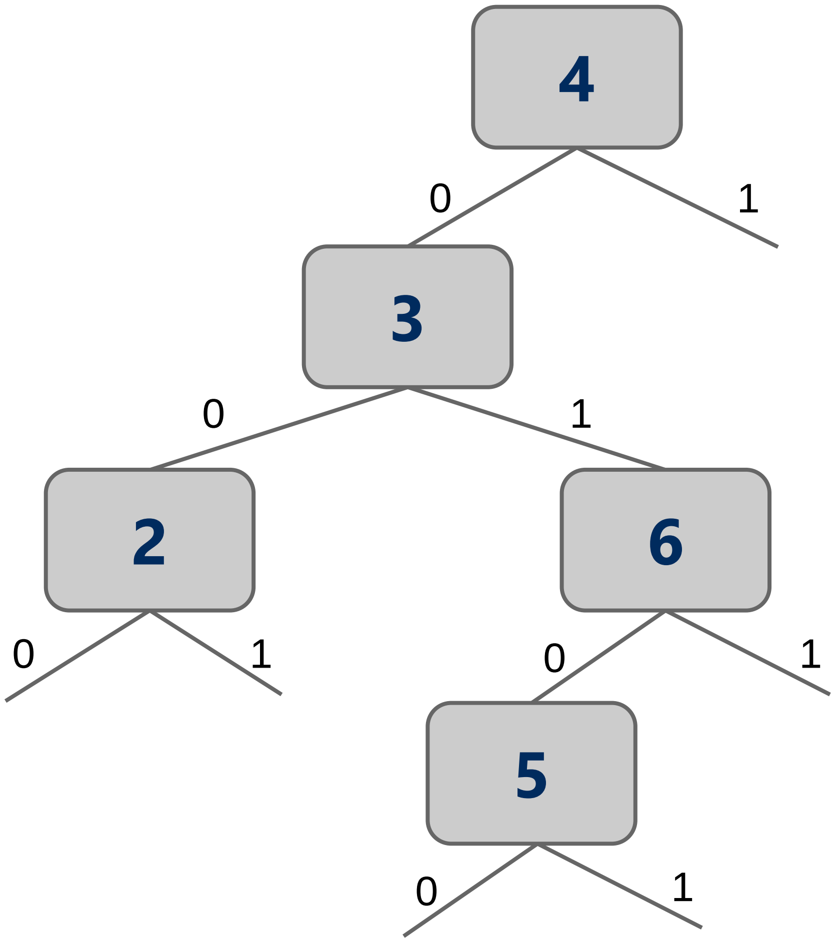
DST and Prefixes

- In a DST, each node shares a **<u>common prefix of length depth(node)</u>** with all nodes in its subtree
  - E.g., 6 shares the prefix "01" with 5
- In-order traversal doesn't produce a sorted order of the items
  - Insertion algorithm can be modified to make a DST a BST at the same time



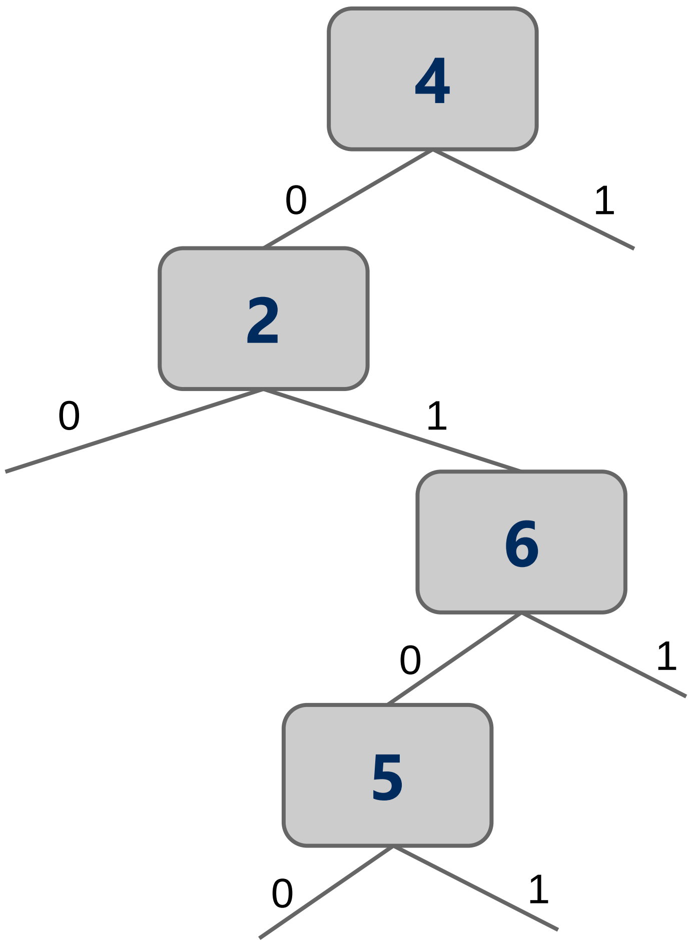
DST example: Delete

- Delete 3
- Can replace it with any leaf in its subtree
- Let's replace it with 2
- OK because 2 shares "0" as a prefix with 3, so it also shares "0" as a prefix with 6 and 5



DST example: Delete

- Delete 3
- Can replace it with any leaf in its subtree
- Let's replace it with 2
- OK because 2 shares "0" as a prefix with 3, so it also shares "0" as a prefix with 6 and 5



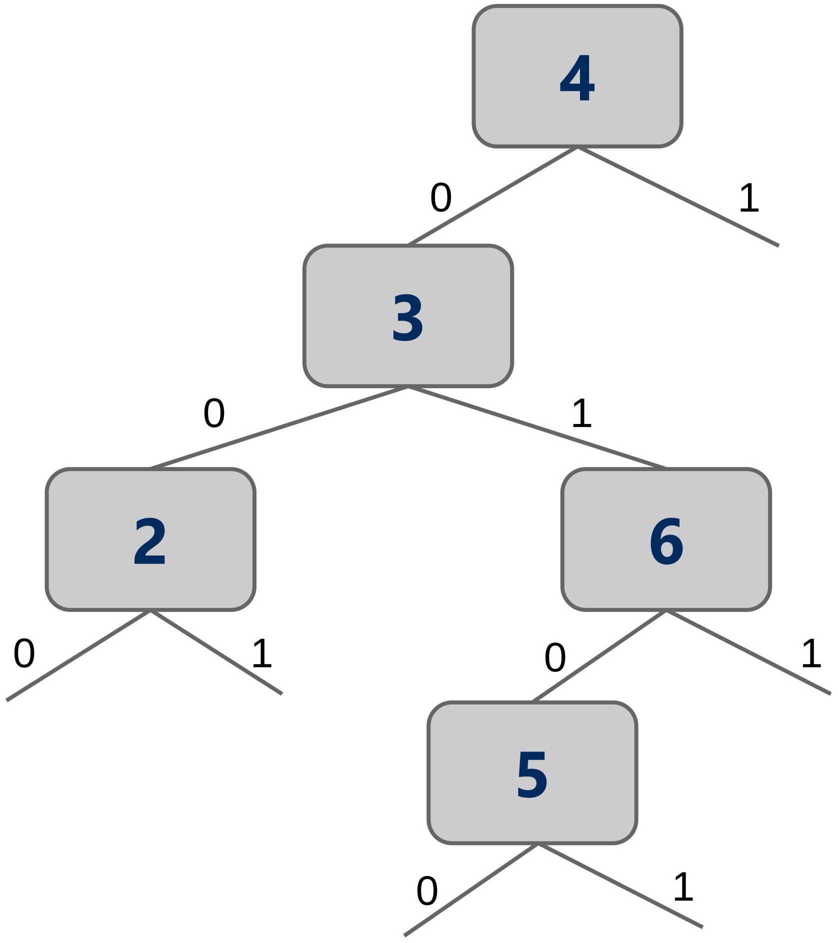
DST example: Variable length keys

- Insert

1 01

- Must be in place of 6

- Replace 6 by 1 and re-insert 6



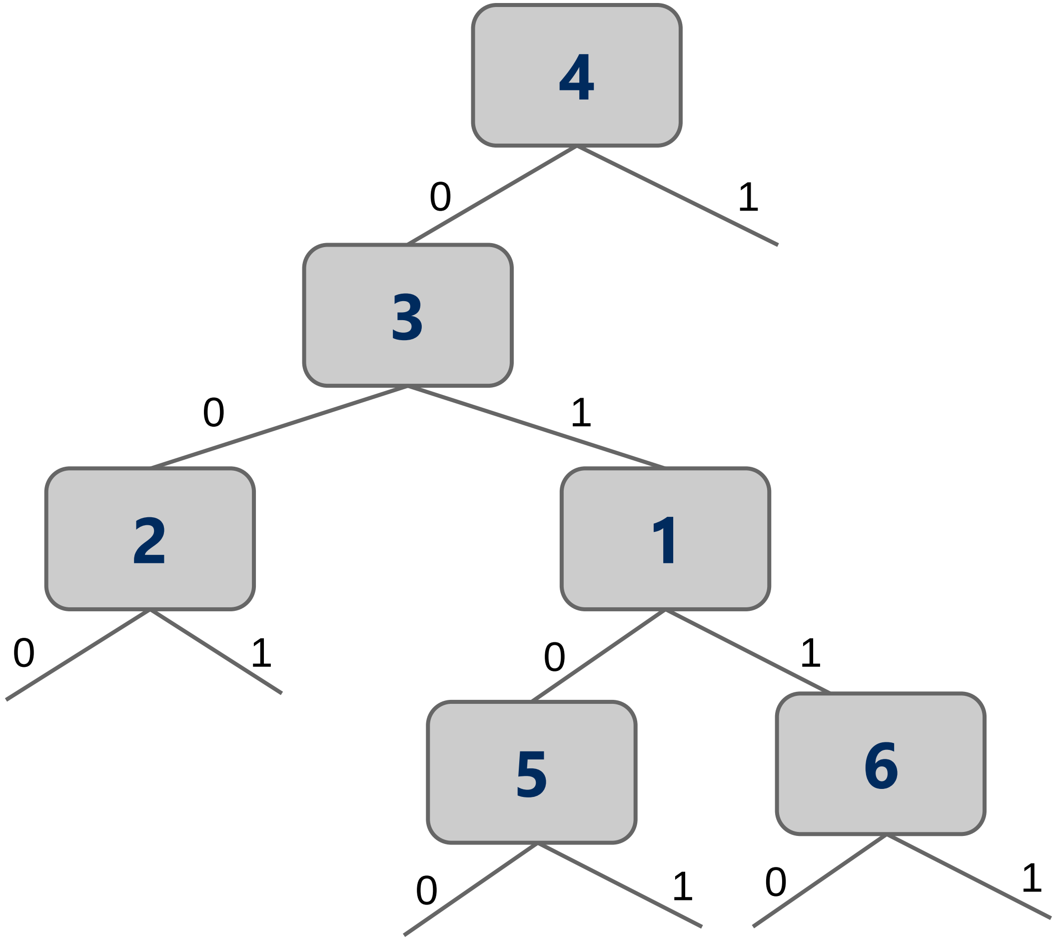
DST example: Variable length keys

- Insert

1 01

- Must be in place of 6
- Replace 6 by 1 and re-insert

6 0110



Analysis of digital search trees

average case runtime $= \sum_{\text{all cases}} Pr(\text{Case}_i) \times$ runtime for Case_i

- Runtime?
  - O(b), b is the bit length of the target or inserted key
  - On average, b = log(n)
  - When branching according to a 0 or 1 is equally likely
  - In general b >= $\lceil \log n \rceil$
- We end up doing many **equality** comparisons against the full key
- This is better than less than/greater than comparison in BST
- Can we improve on this?

Radix search tries (RSTs)

- Trie as in retrieve, pronounced the same as "try"
- Instead of storing keys inside nodes in the tree, we store them implicitly as paths down the tree
  - Interior nodes of the tree only serve to direct us according to the bitstring of the key
  - Values can then be stored at the end of key's bitstring path (i.e., at leaves)
  - RST uses less space than BST and DST

Adding to Radix Search Trie (RST)

- Input: key and corresponding value
- if root is null, set root ← new node
- current node ← root
- for each *bit* in the key
  - if bit == 0,
    - if left child of current node is null, create a new node and attach as the left child
    - move to left child
      - either recursively or by setting current ← current.left
  - if bit == 1,
    - if right child of current node is null, create a new node and attach as the right child
    - move to right child
      - either recursively or by setting current ← current.right
- insert corresponding value into current node

RST example

Insert:

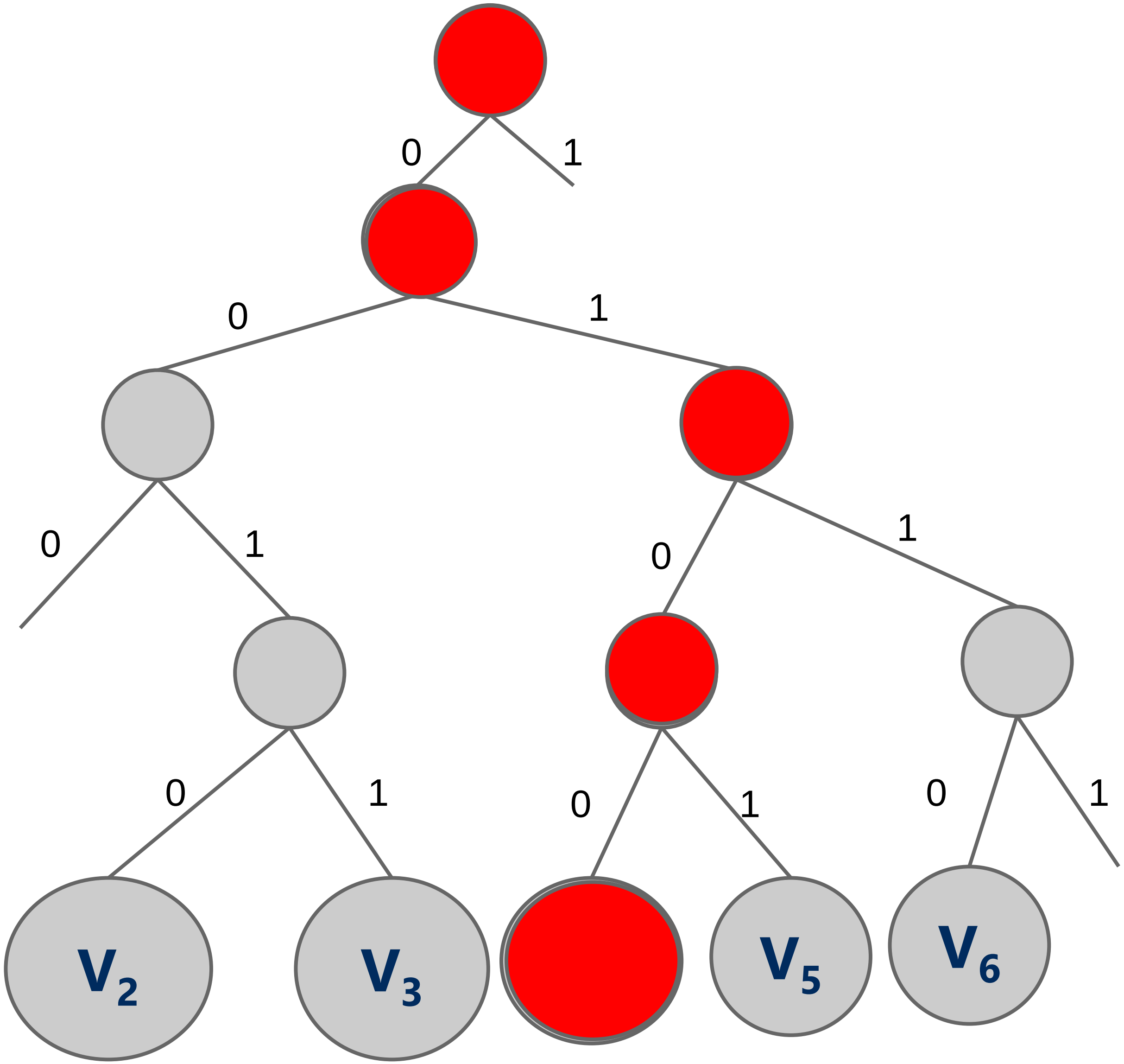
4 0100

3 0011

2 0010

6 0110

5 0101



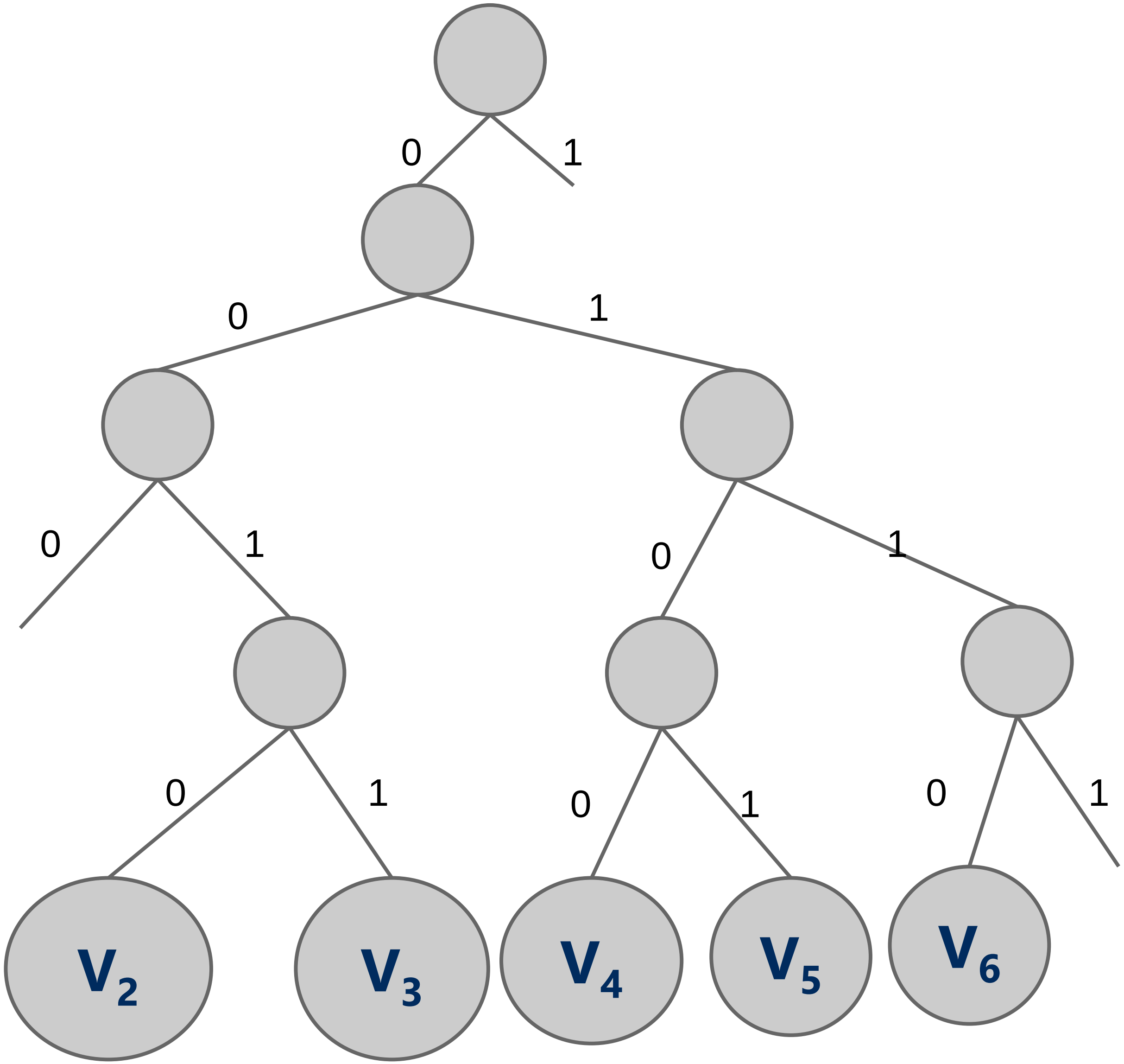
Searching in Radix Search Trie (RST)

- Input: key
- current node ← root
- for each *bit* in the key
  - if current node is null, return *key not found*
  - if bit == 0,
    - move to left child
      - either recursively or by setting current ← current.left
  - if bit == 1,
    - move to right child
      - either recursively or by setting current ← current.right
- if current node is null or the value inside is null
  - return key *not found*
- else return the value stored in current node

RST example

Search:

3 0011

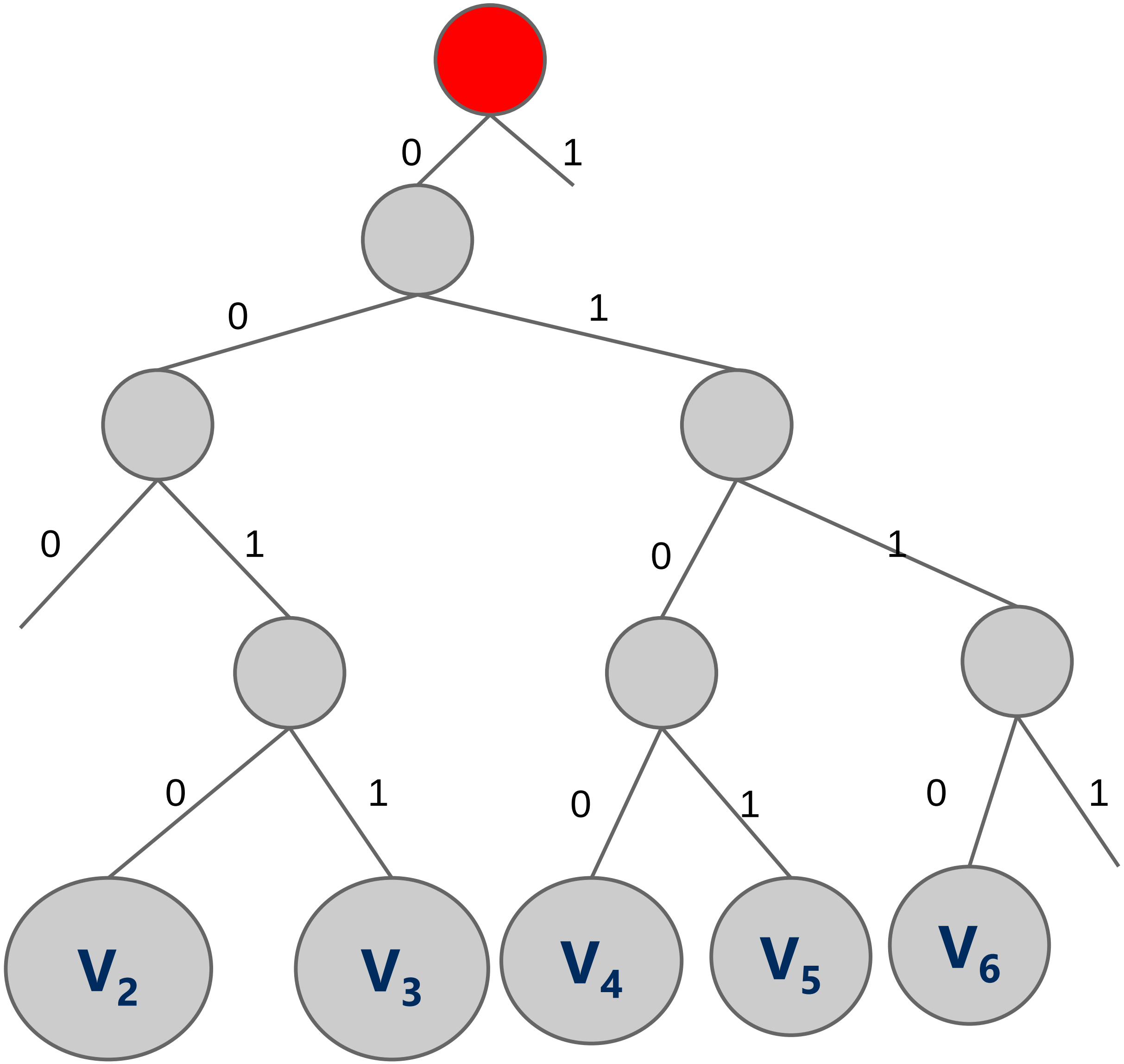


RST example

Search:

3 0011

7 0111

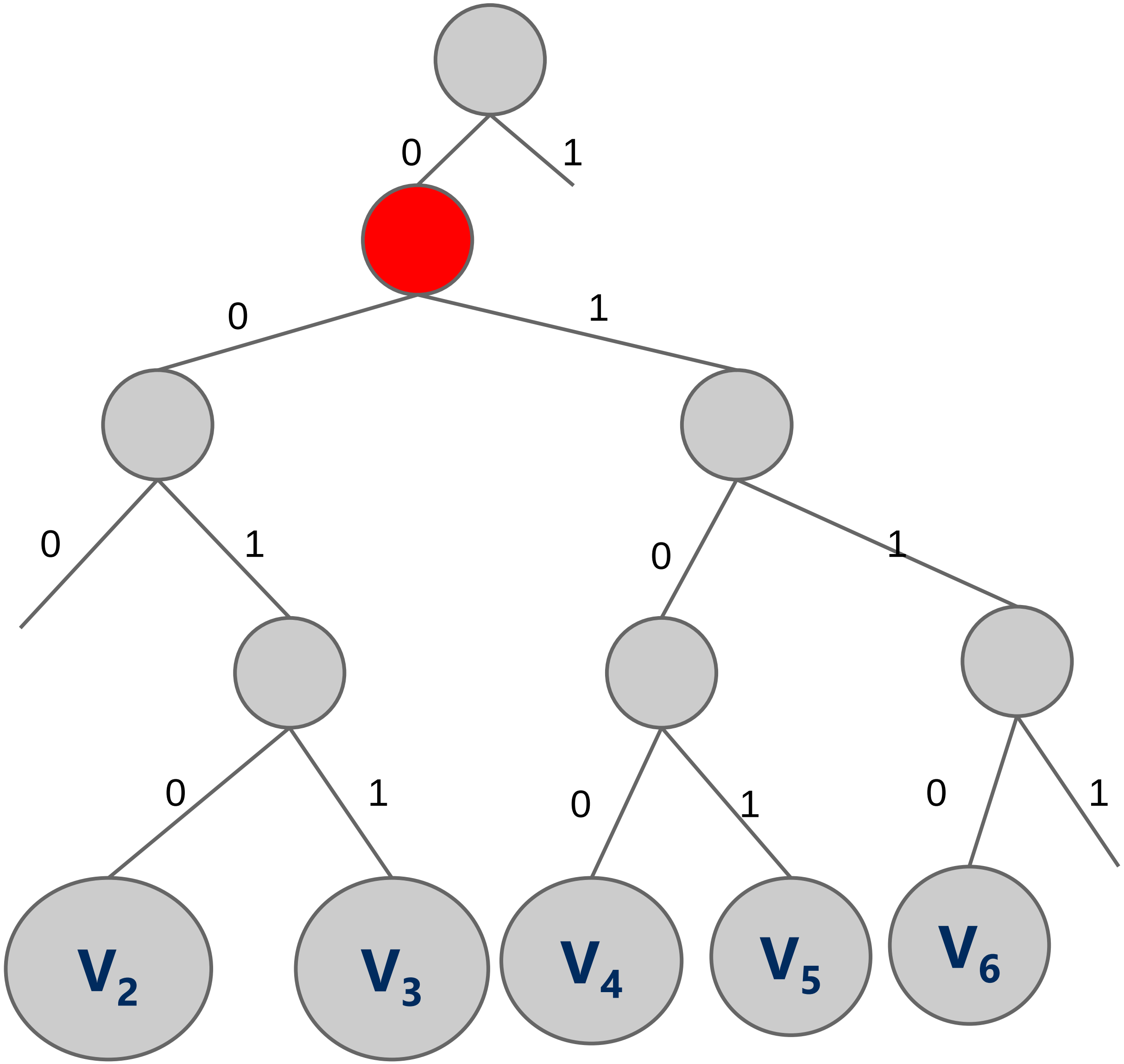


RST example

Search:

3 0011

7 0111

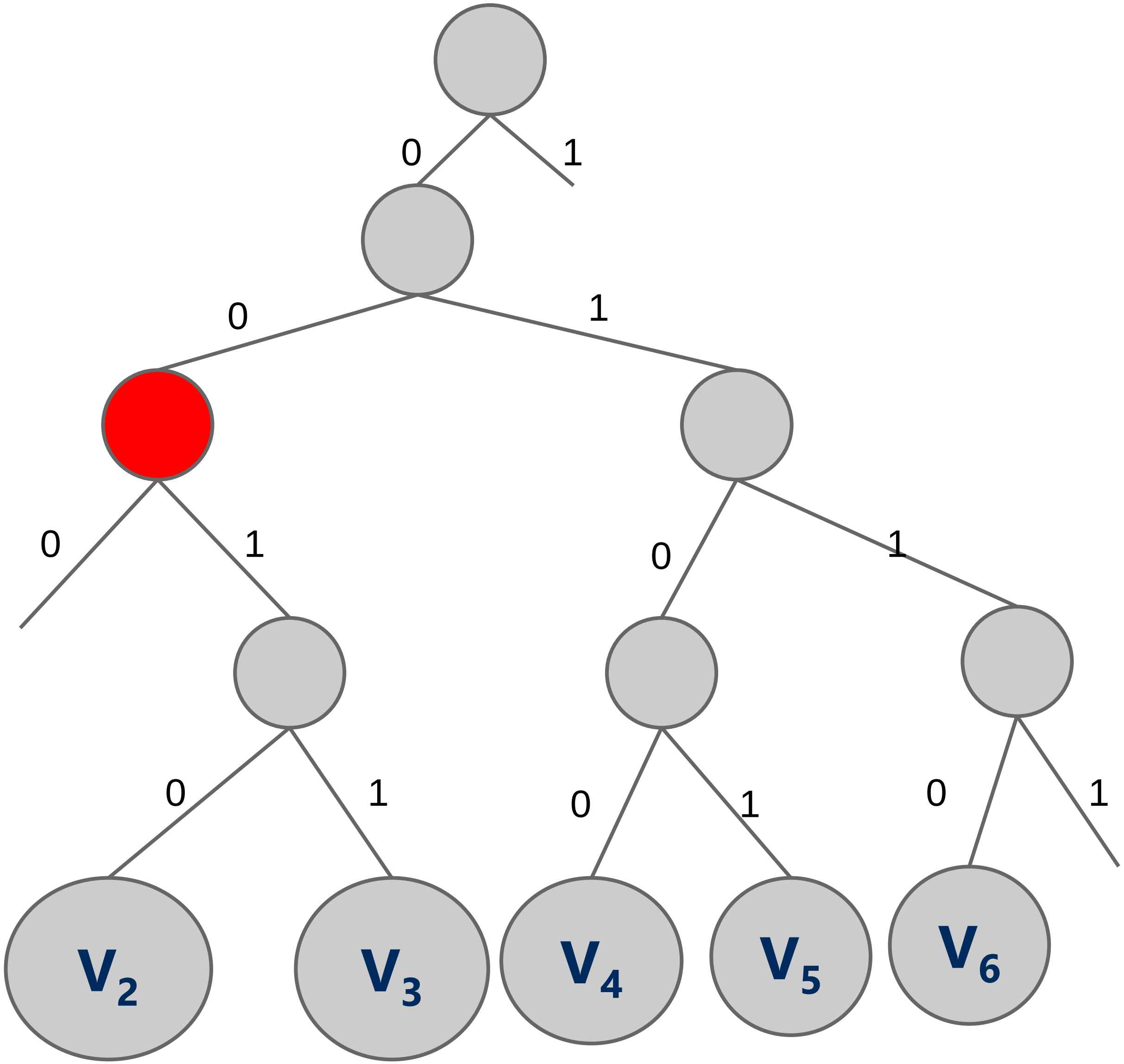


RST example

Search:

3 0011

7 0111

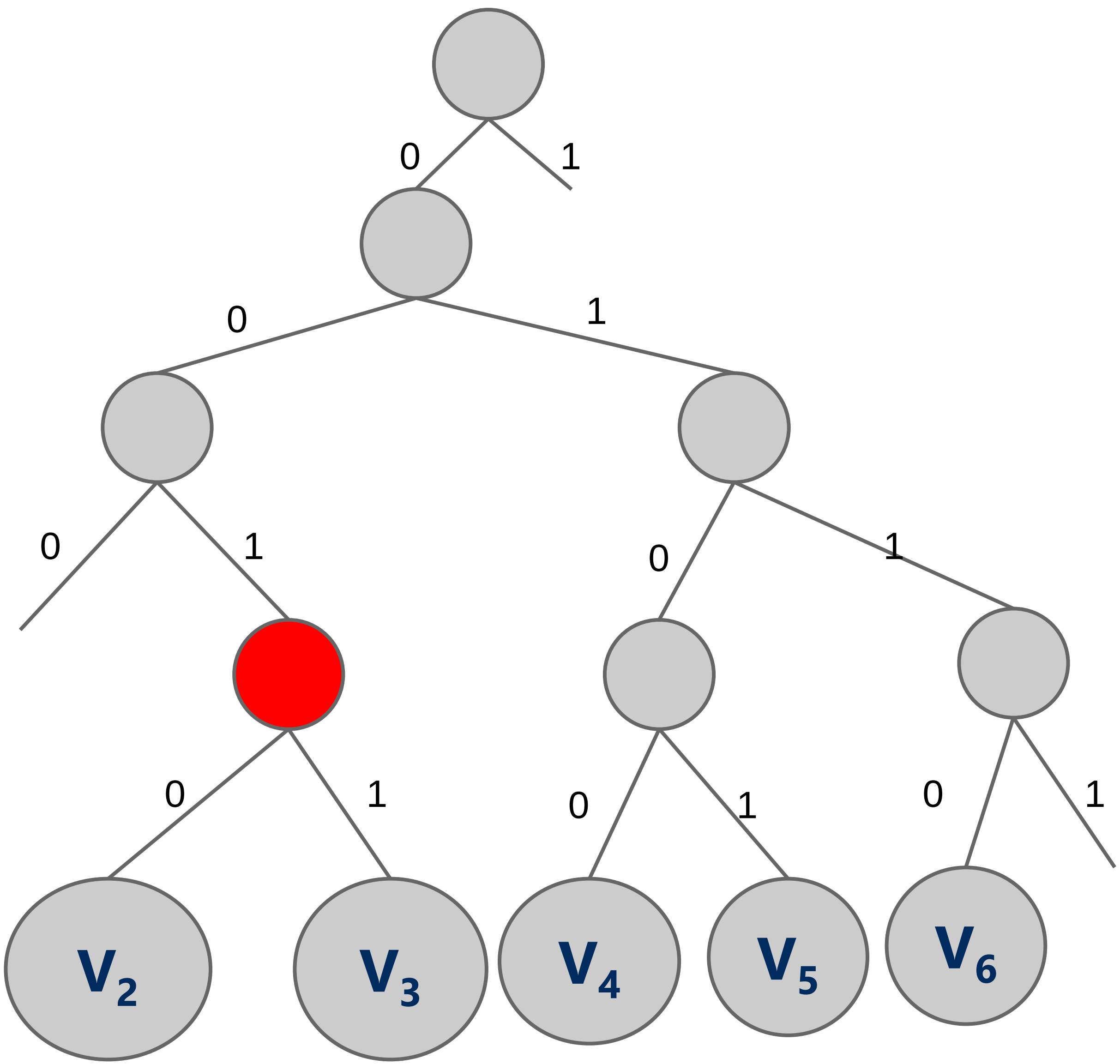


RST example

Search:

3 0011

7 0111

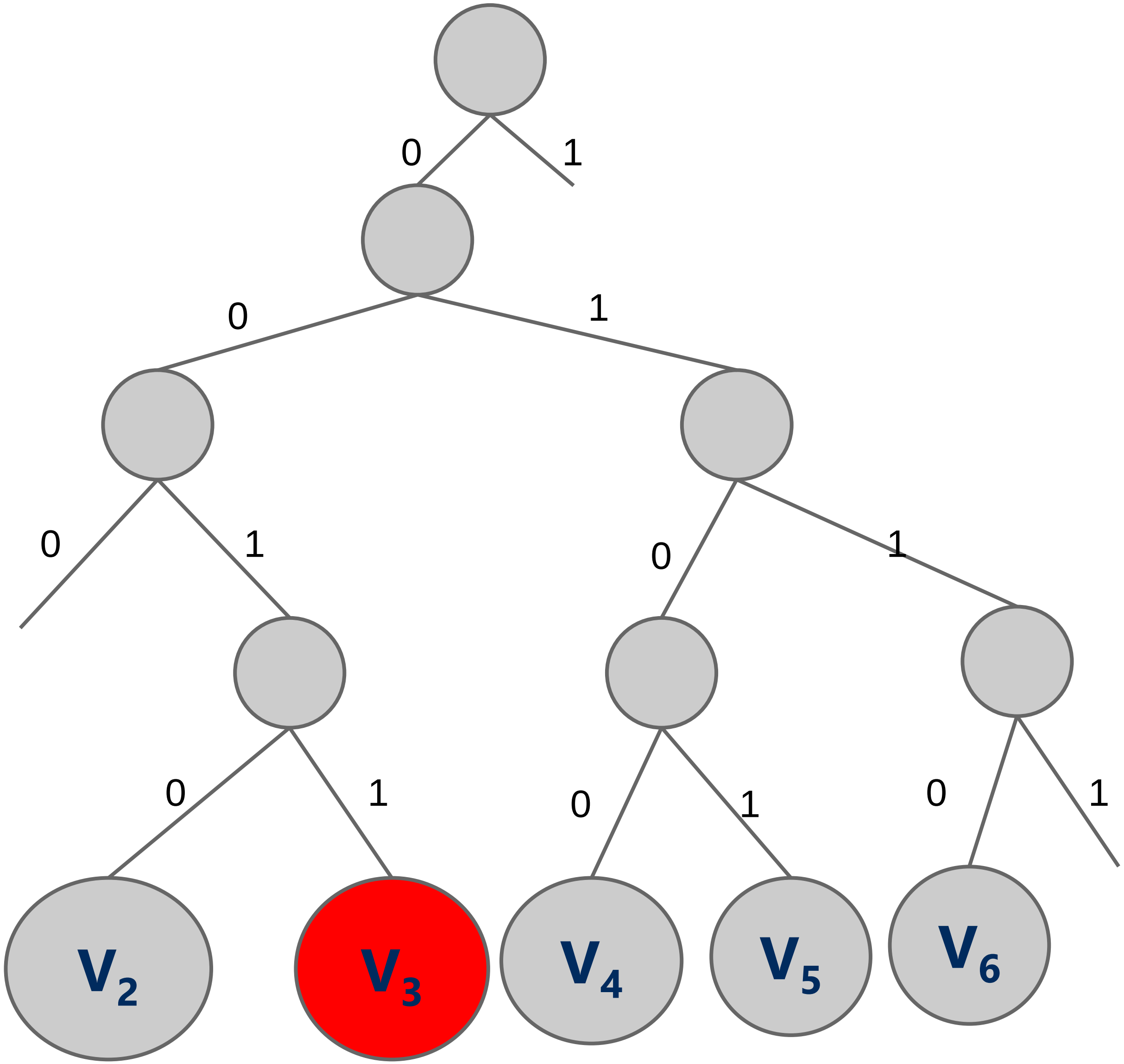


RST example

Search:

3 0011

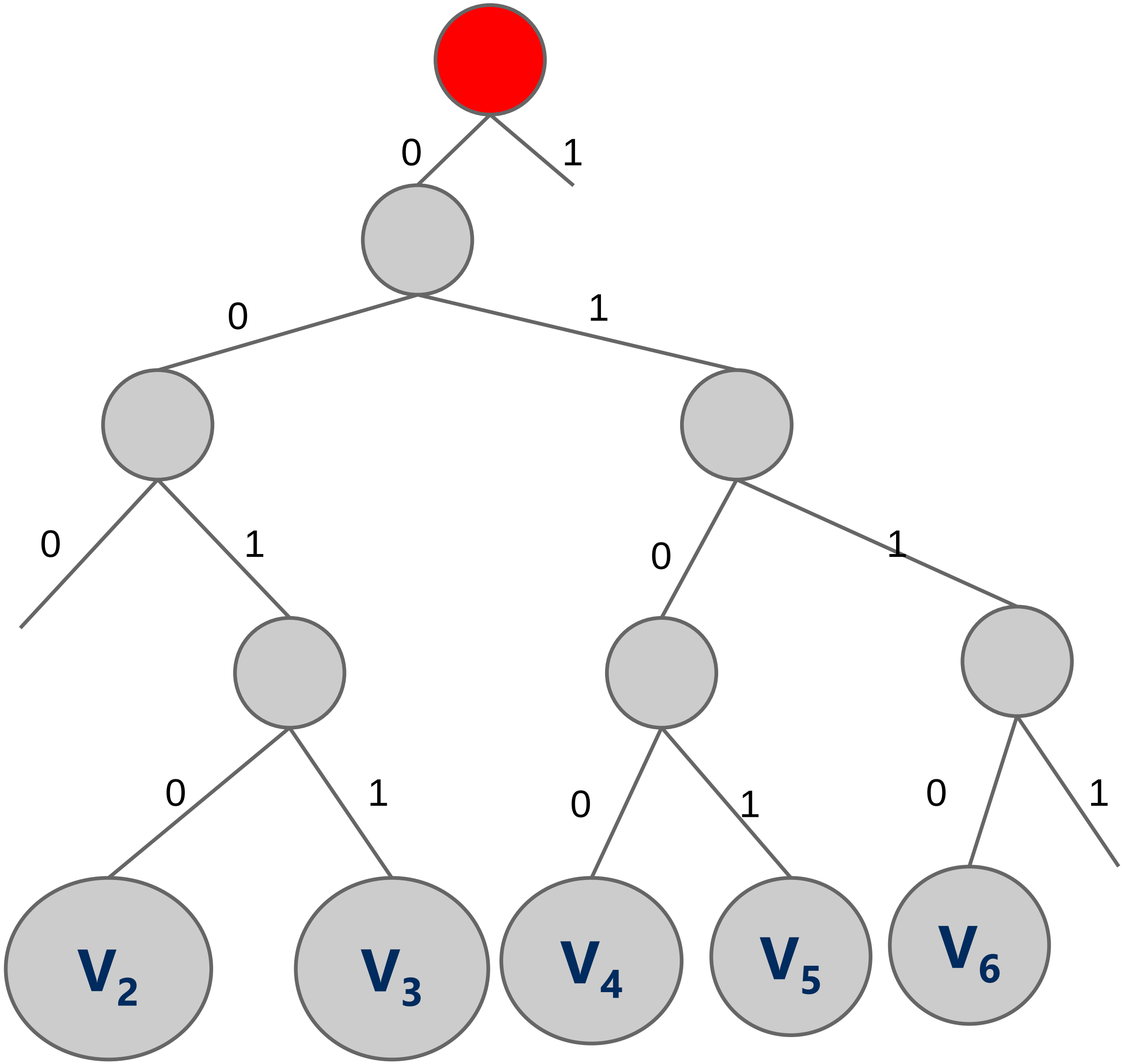
7 0111



RST example

Search:

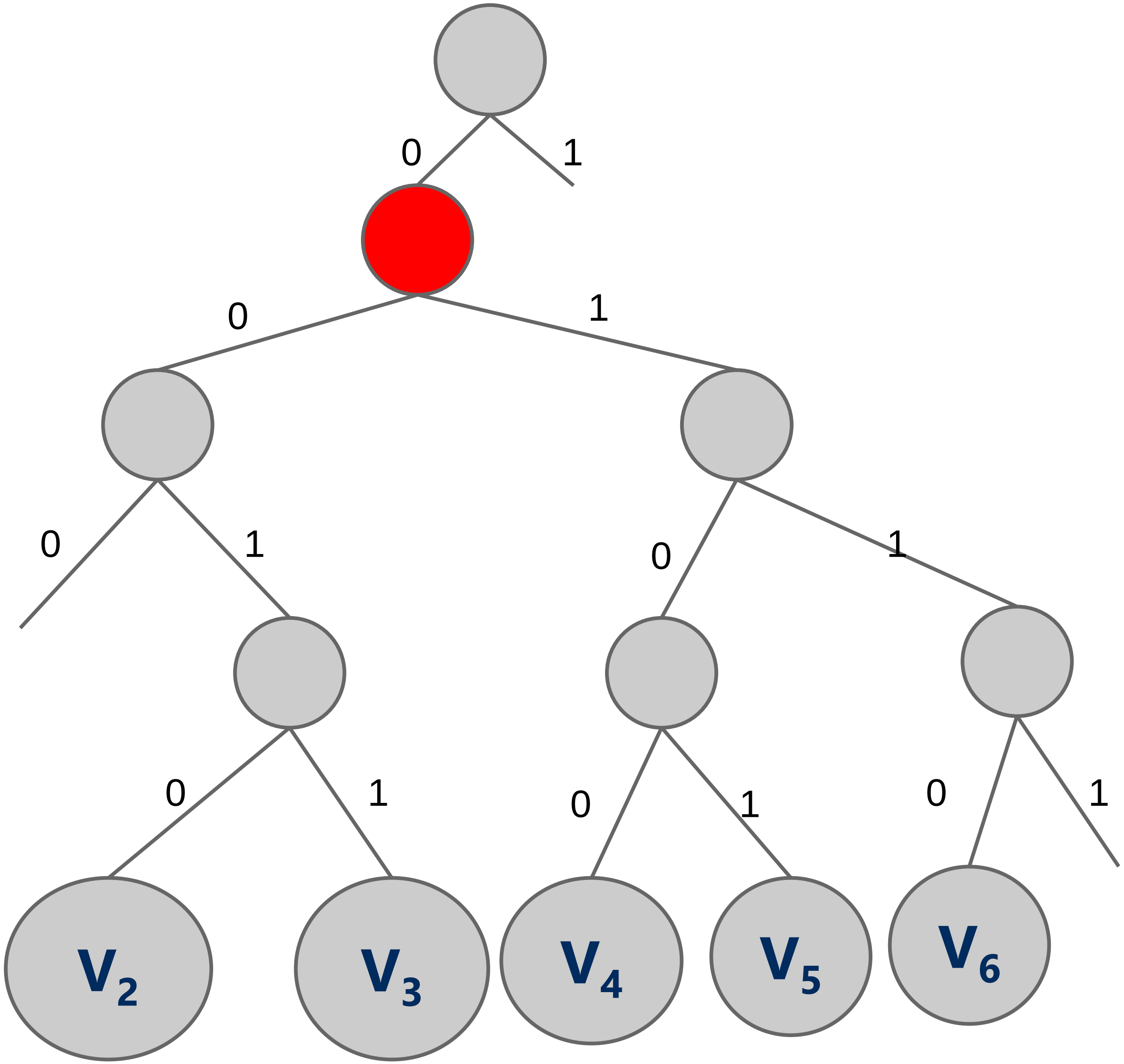
7 0111



RST example

Search:

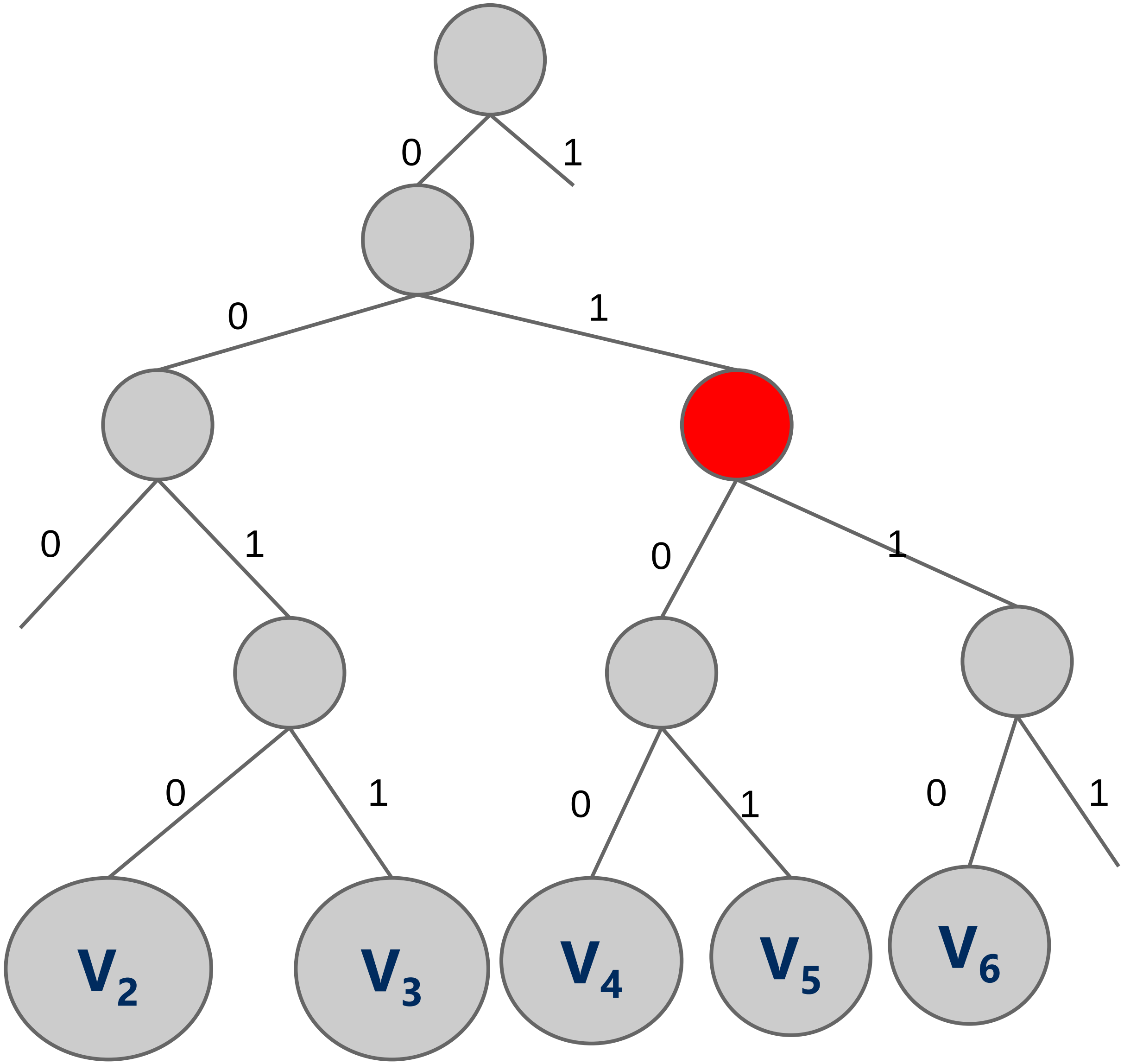
7 0111



RST example

Search:

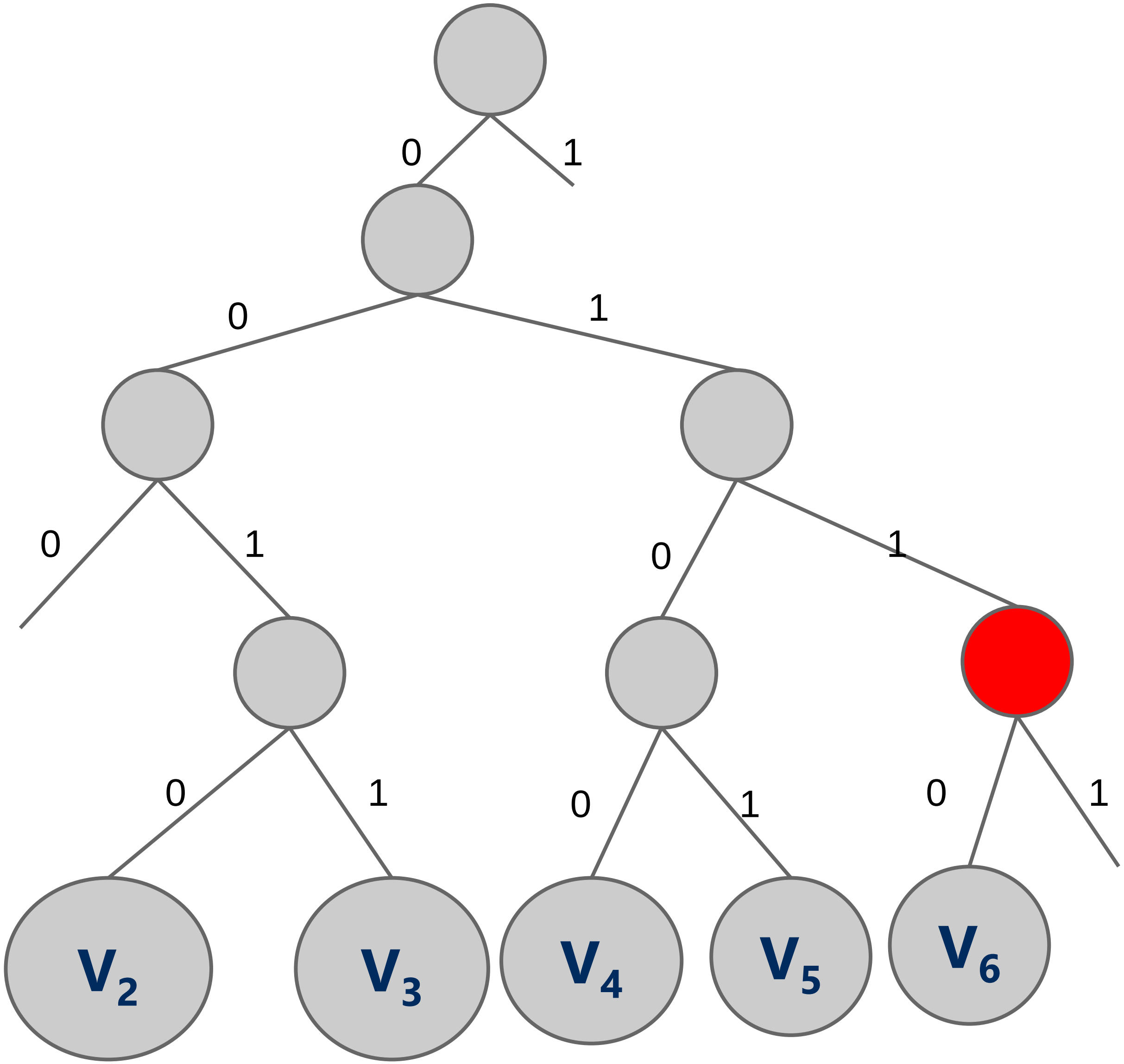
7 0111



RST example

Search:

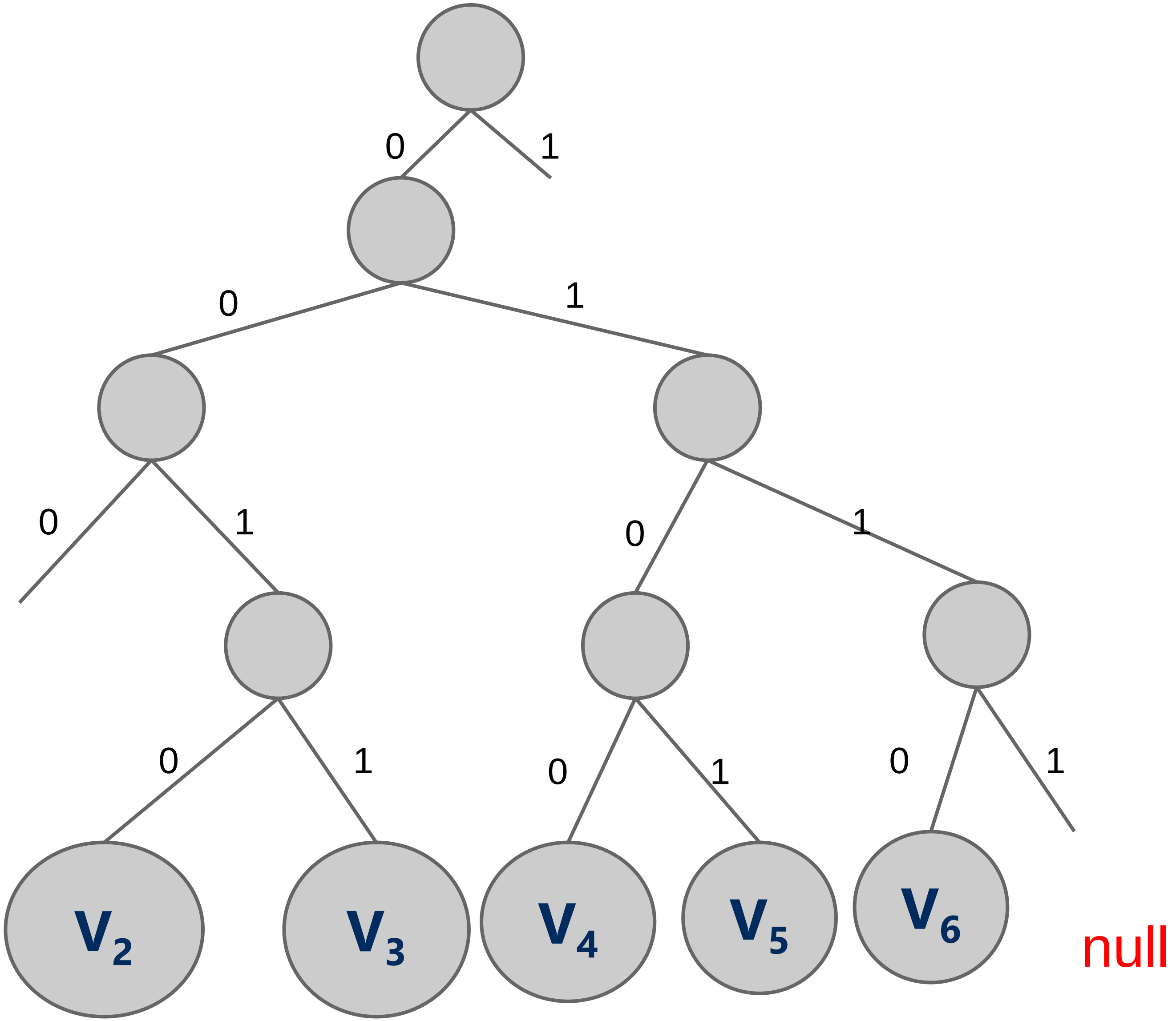
7 0111



RST example

Search:

7 0111



RST analysis

- Runtime?
- O(b), the bit length of the key
  - However, this time we don't have full key comparisons
- Would this structure work as well for other key data types?
- Characters?
  - Characters are the same as 8-bit ints (assuming simple ascii)
- Strings?
- May have huge bit lengths
- How to store Strings?

Larger branching factor tries

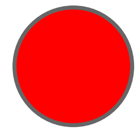
- In our binary-based Radix search trie, we considered one bit at a time
- What if we applied the same method to characters instead of bits in a string?
    - What would this new structure look like?
    - How many children per node?
        - up to R (the alphabet size)
    - Also called R-way radix search tries

Adding to R-way Radix RST

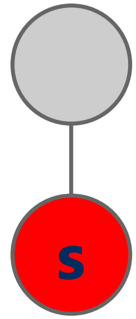
- if root is null, set root ← new node
- current node ← root
- for each *character c* in the key
  - Find the c*th* child
    - if child is null, create a new node and attach as the c*th* child
    - move to child
      - either recursively or by current ← child
- if at last character of key, insert value into current node

Another trie example

she

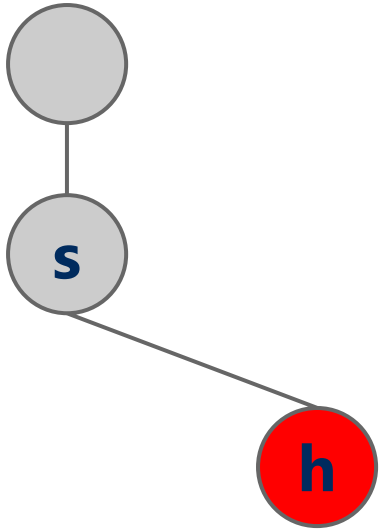


Another trie example



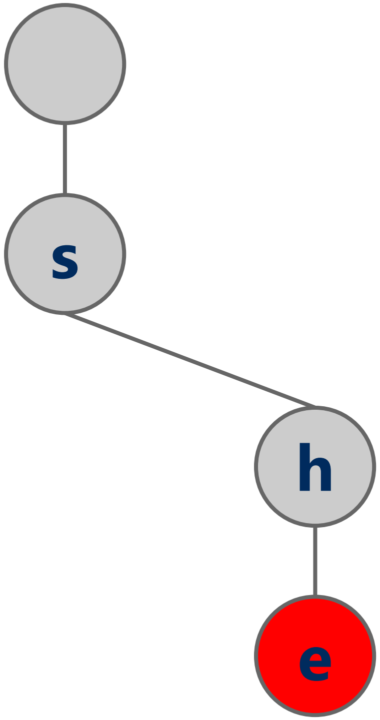
she

Another trie example



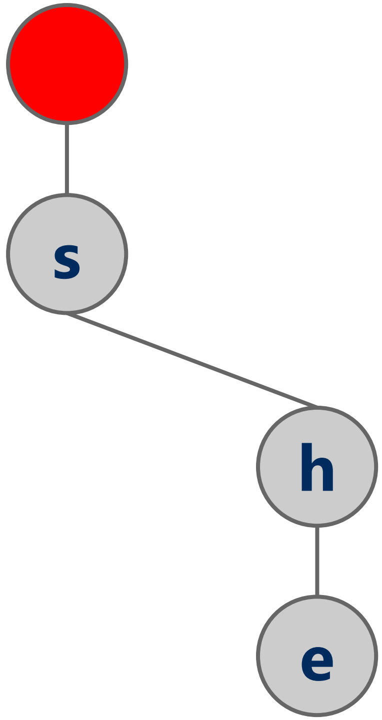
she

Another trie example



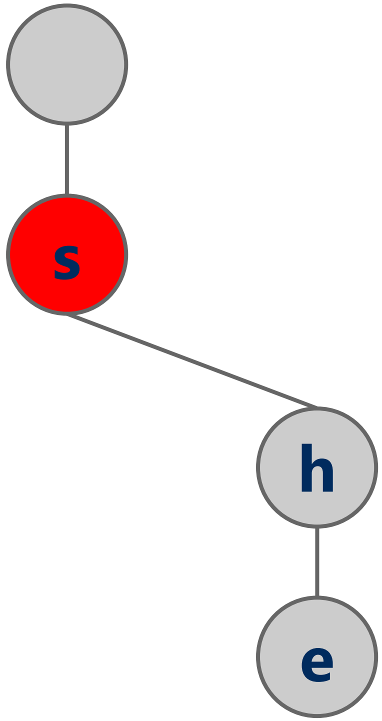
she

Another trie example



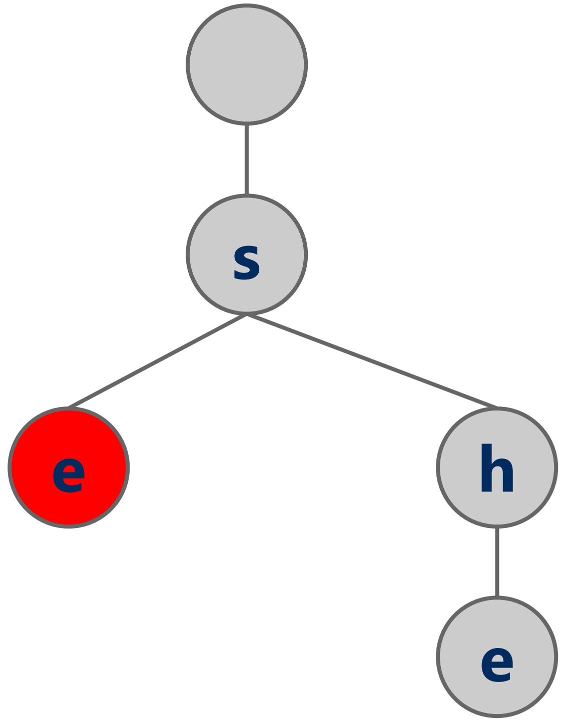
sells

Another trie example



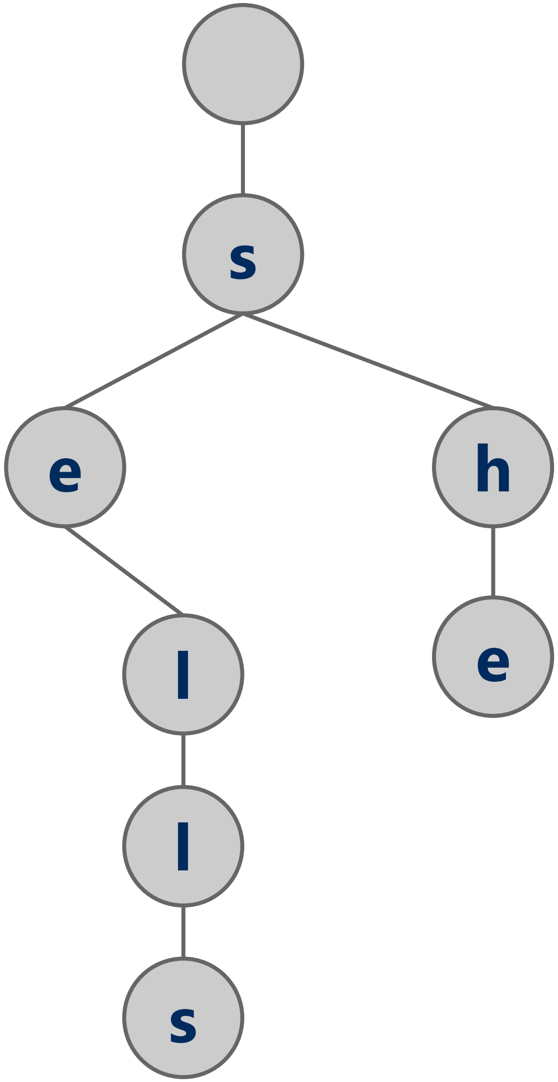
sells

Another trie example



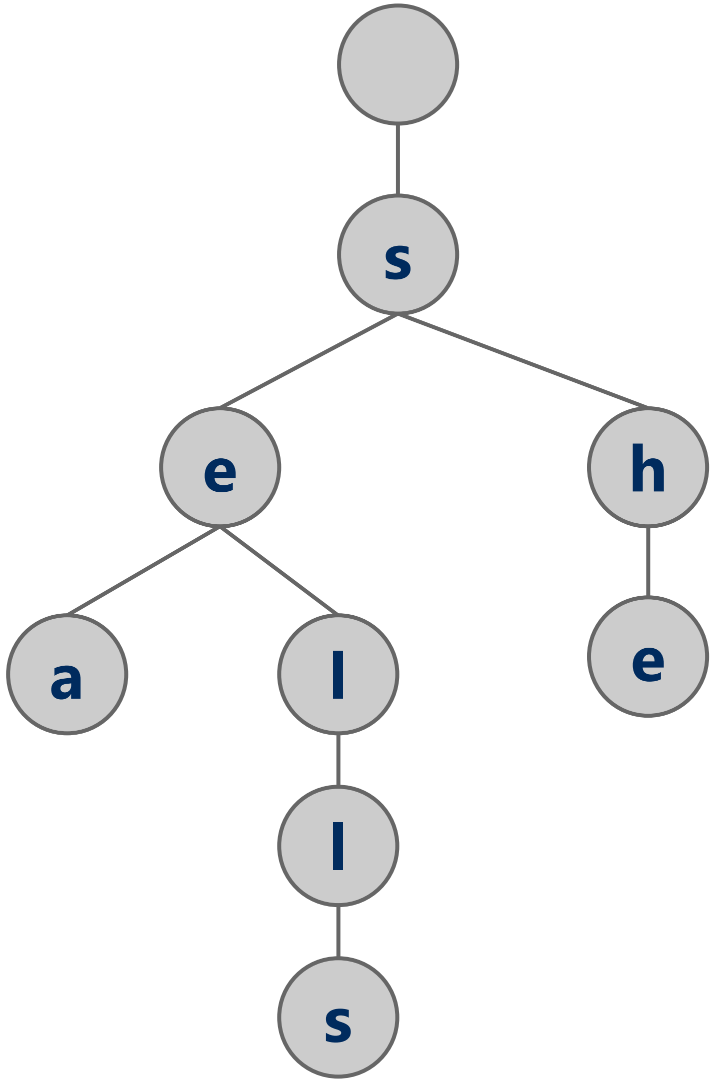
sells

Another trie example



sells

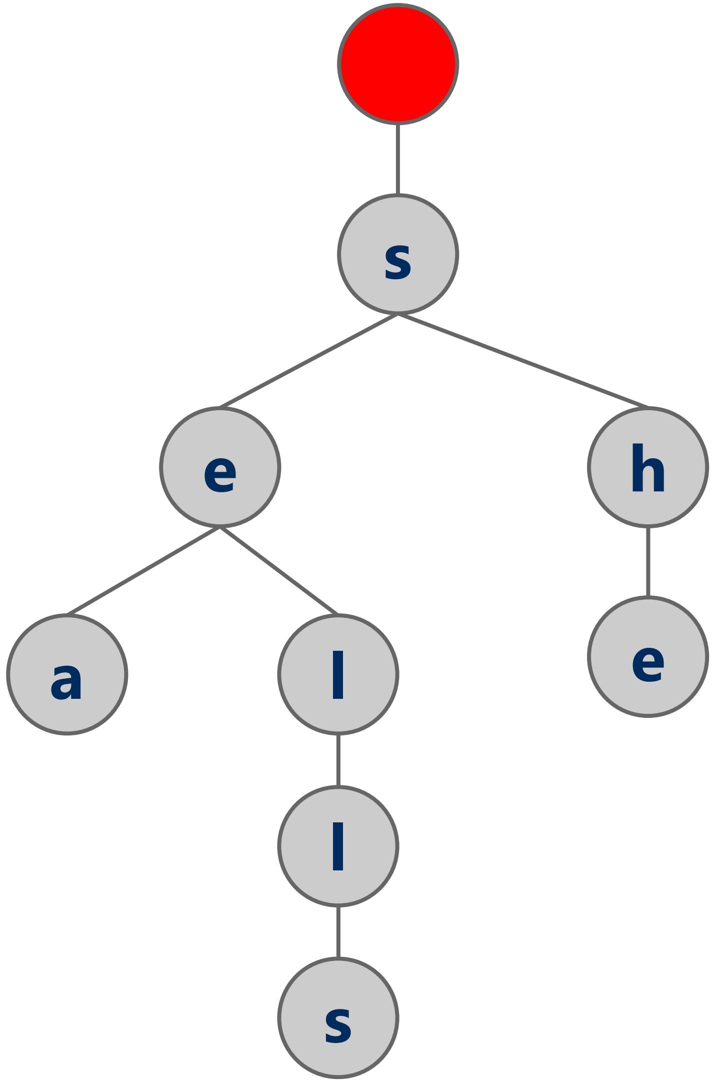
Another trie example



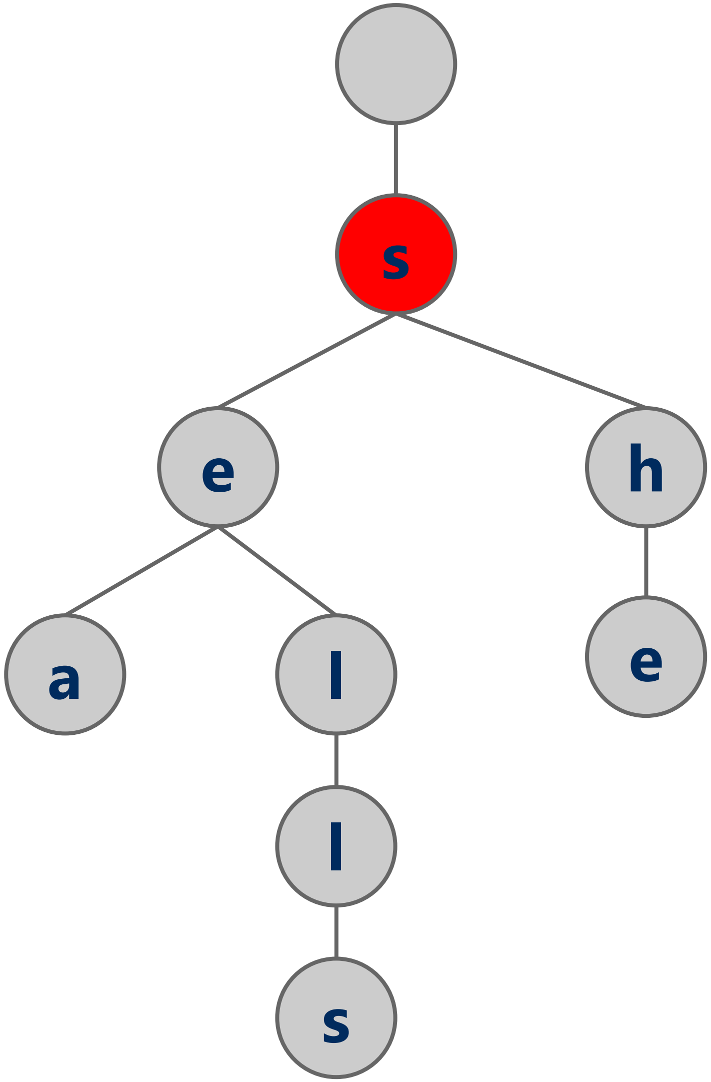
sea

Another trie example

shells

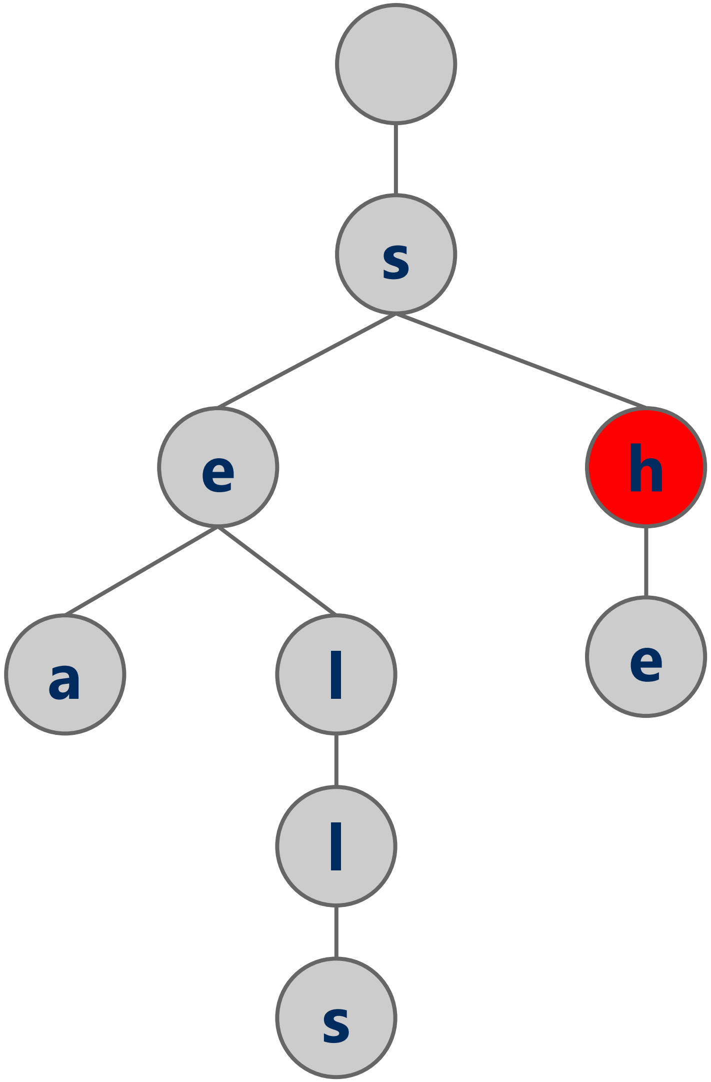


Another trie example



shells

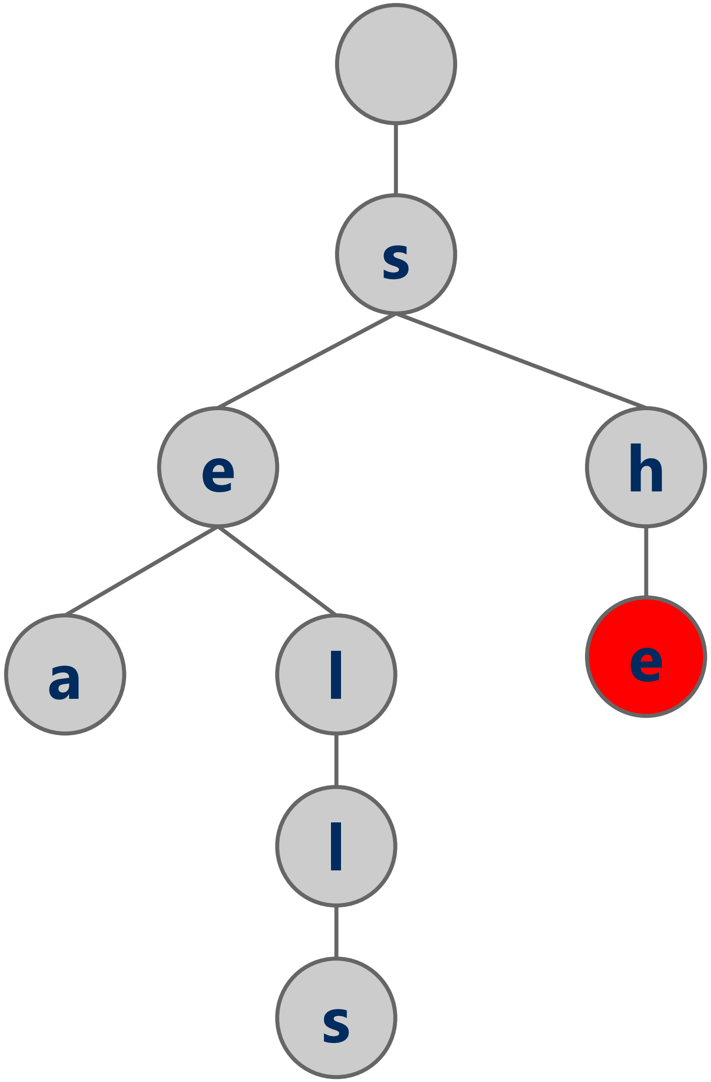
Another trie example



shells

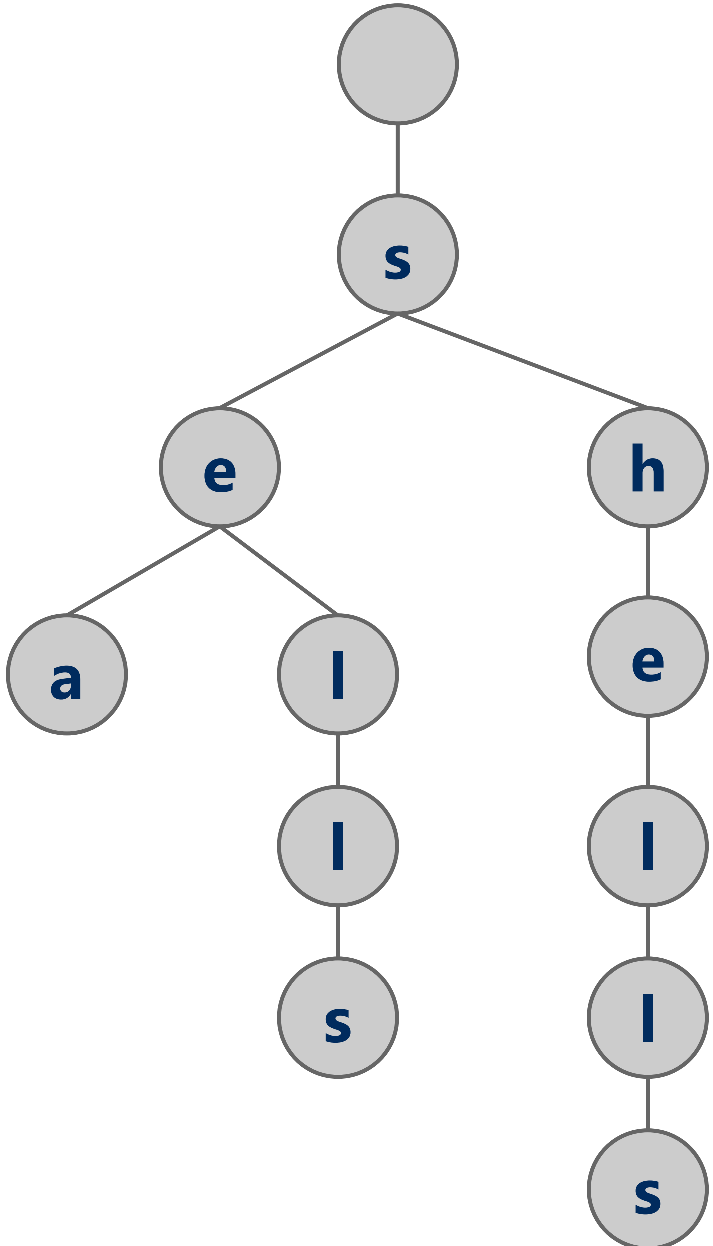
Another trie example

shells

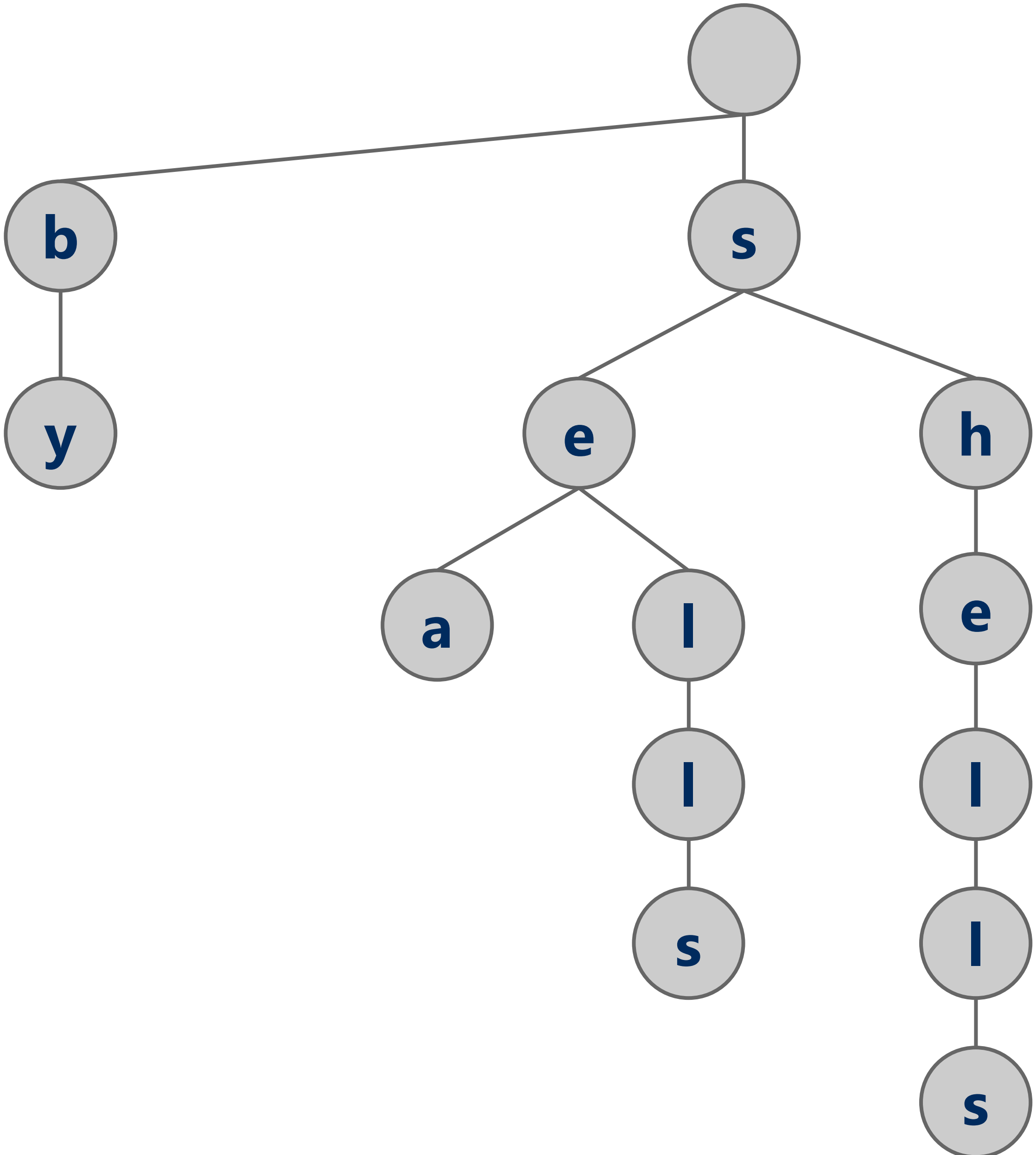


Another trie example

shells

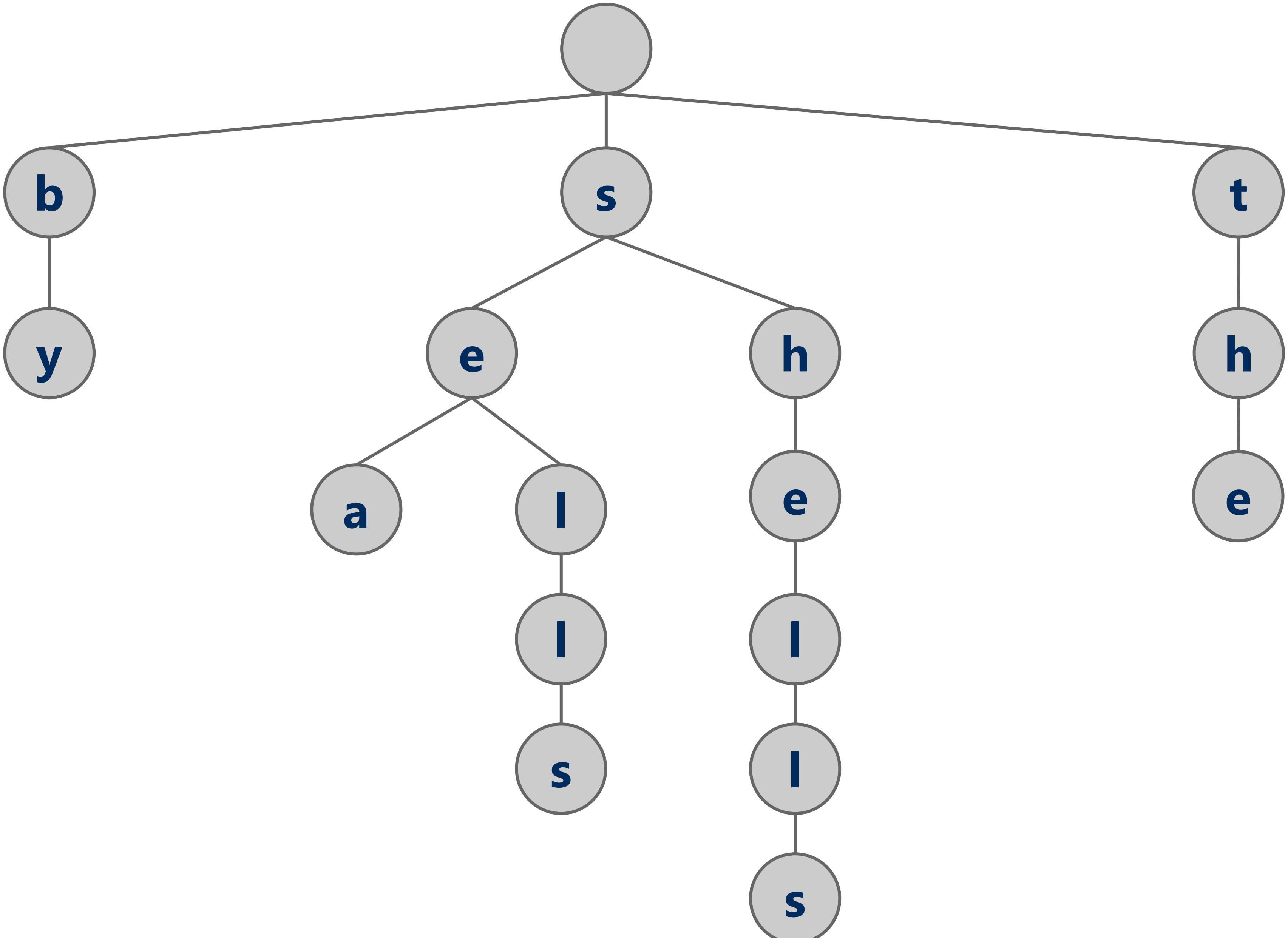


Another trie example



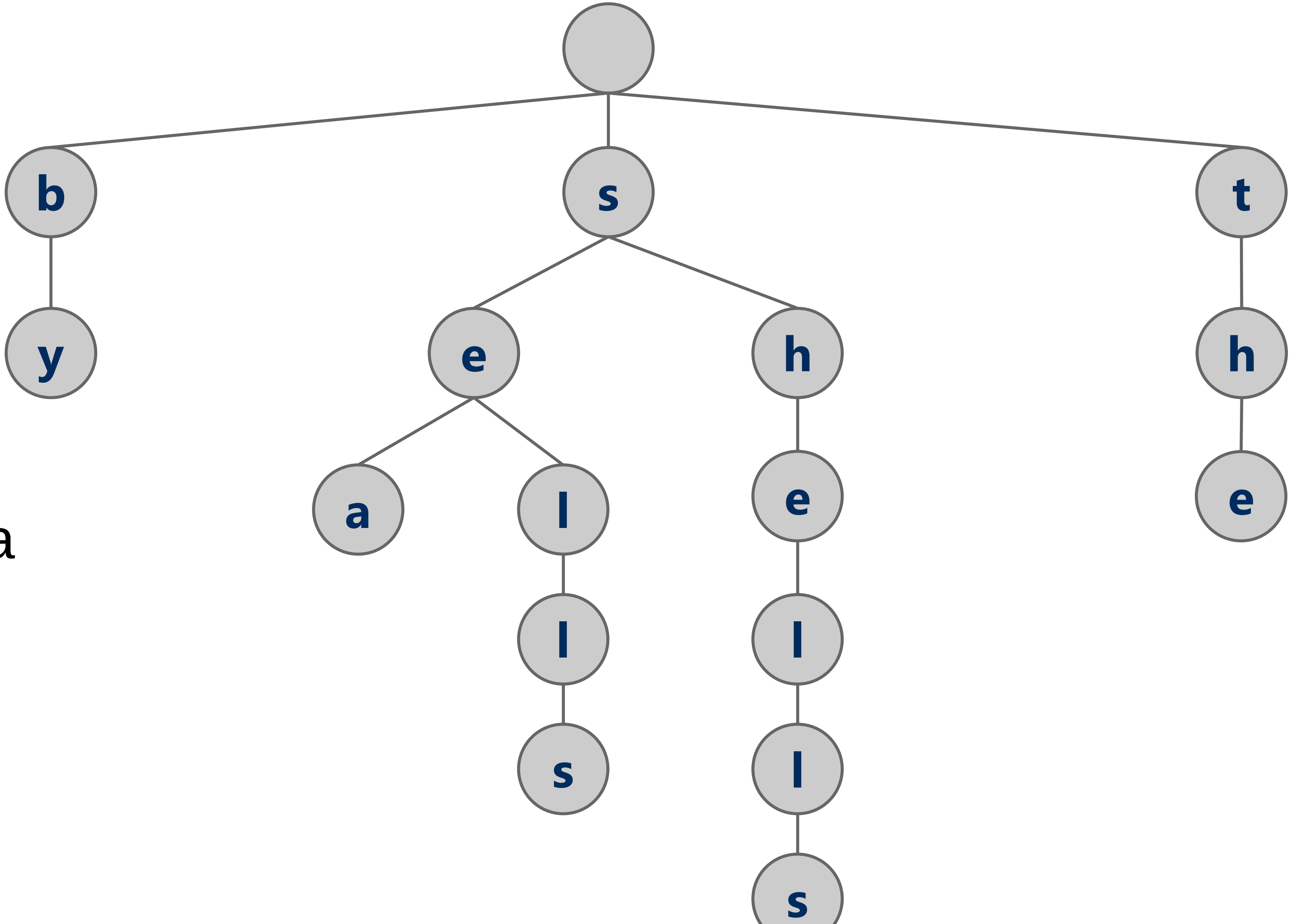
by

Another trie example



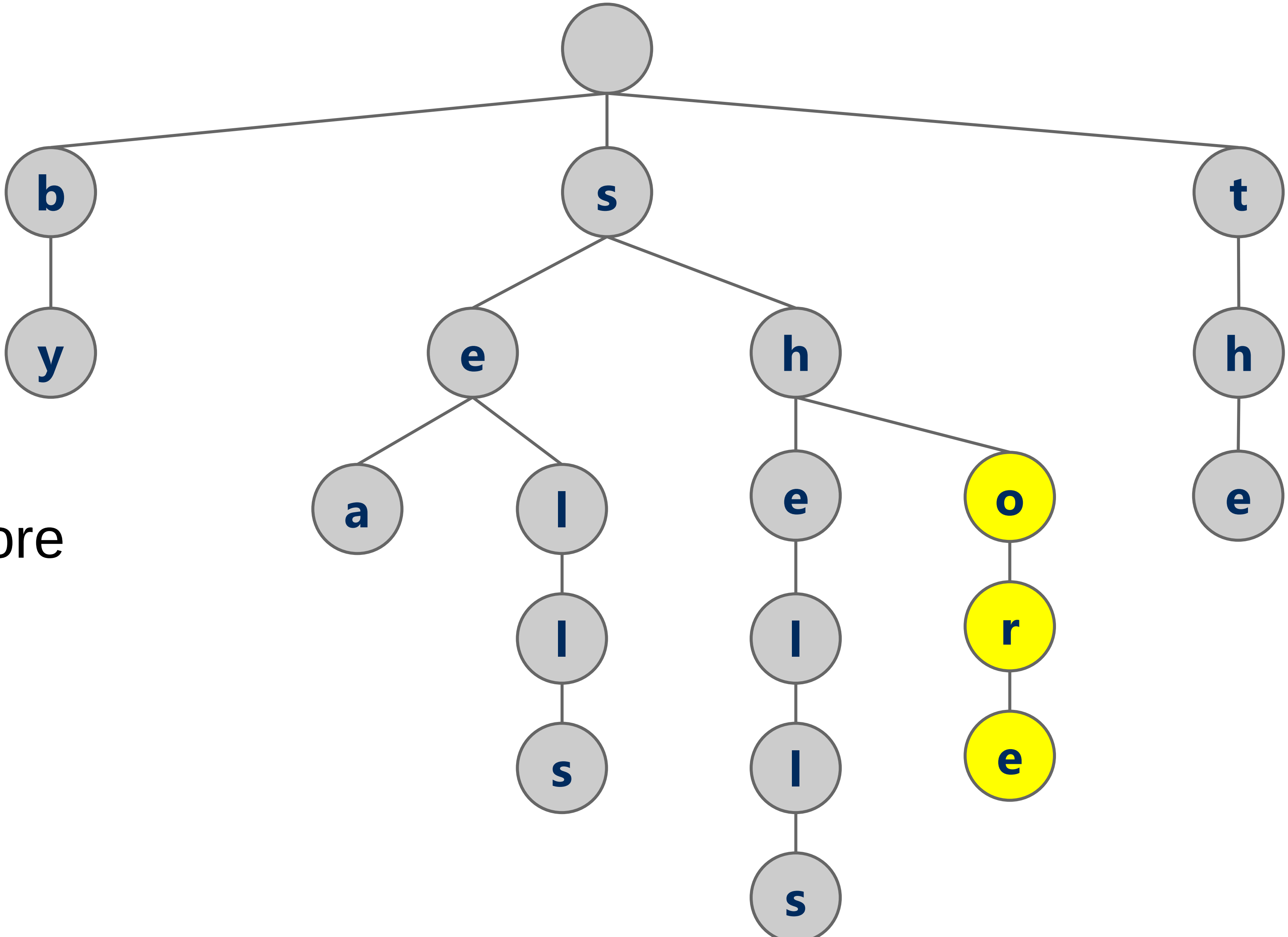
the

Another trie example



sea

Another trie example



shore

Analysis

- Runtime of add and *search hit*?
- O(w) where w is the character length of the string
  - So, what do we gain over RSTs?
    - $w < b$
    - e.g., assuming fixed-size encoding $$w = \frac{b}{\lceil \log R \rceil}$$
    - tree height is reduced

Search Miss

- Search Miss time for R-way RST
  - Require an average of $log_R(n)$ nodes to be examined
    - Proof in Proposition H of Section 5.2 of the text
- Average tree height with 2^{20} keys in an RST?
  - $\log_2 n = \log_2 2^{20} = 20$
- With 2^{20} keys in a large branching factor trie, assuming 8-bits at a time?
  - $\log_R n = \log_{256} 2^{20} = \log_{256} (2^8)^{2.5} = = \log_{256} 256^{2.5} = 2.5$

Implementation Concerns

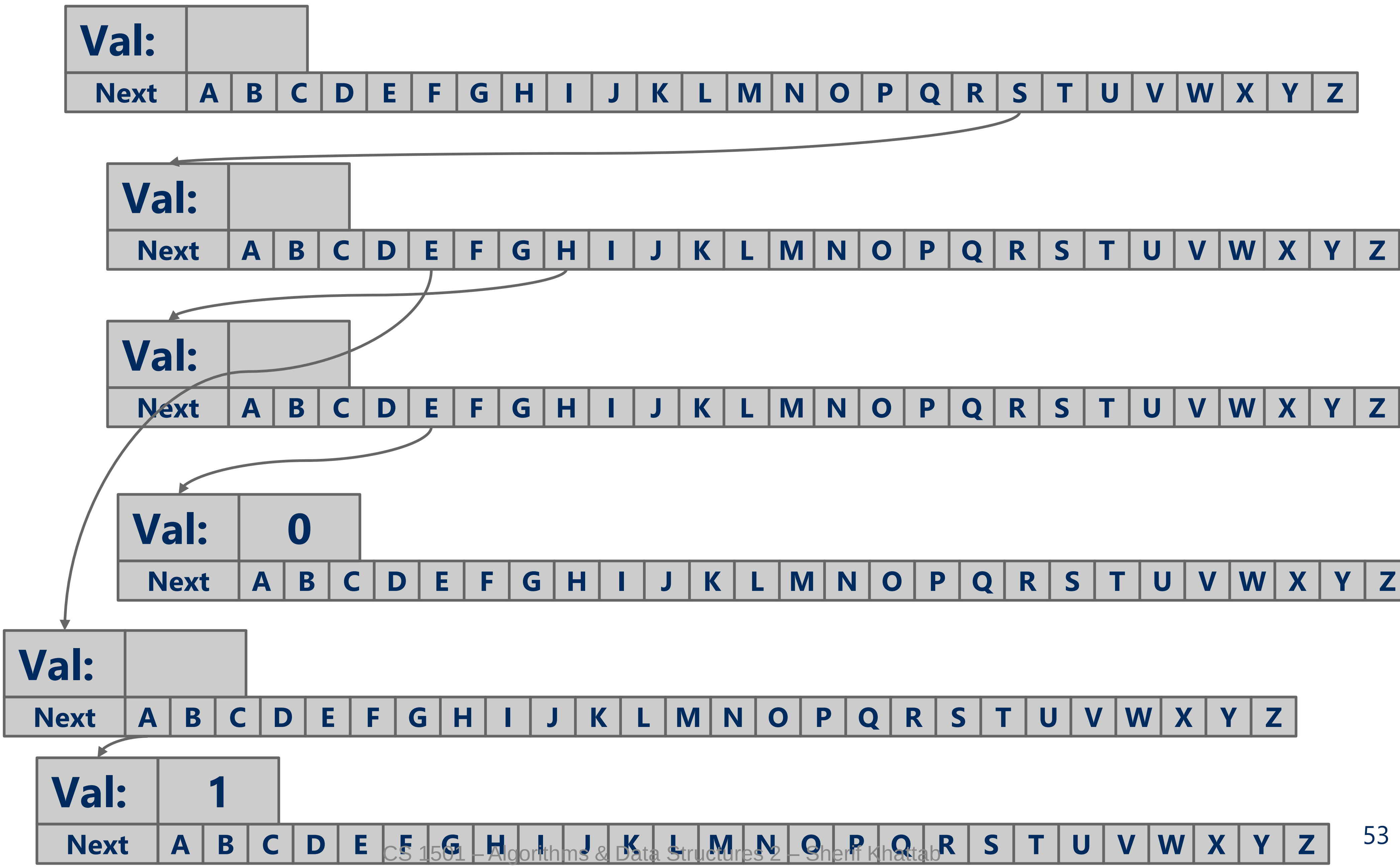
- See TrieSt.java
  - Implements an R-way trie
- Basic node object:

Where R is the branching factor

```
private class Node {
    private Object val;
    private Node[] next;
    private Node(){
      next = new Node[R];
    }
}
```

- Non-null **val** means we have traversed to a valid key
- Again, note that keys are not directly stored in the trie at all

R-way trie example

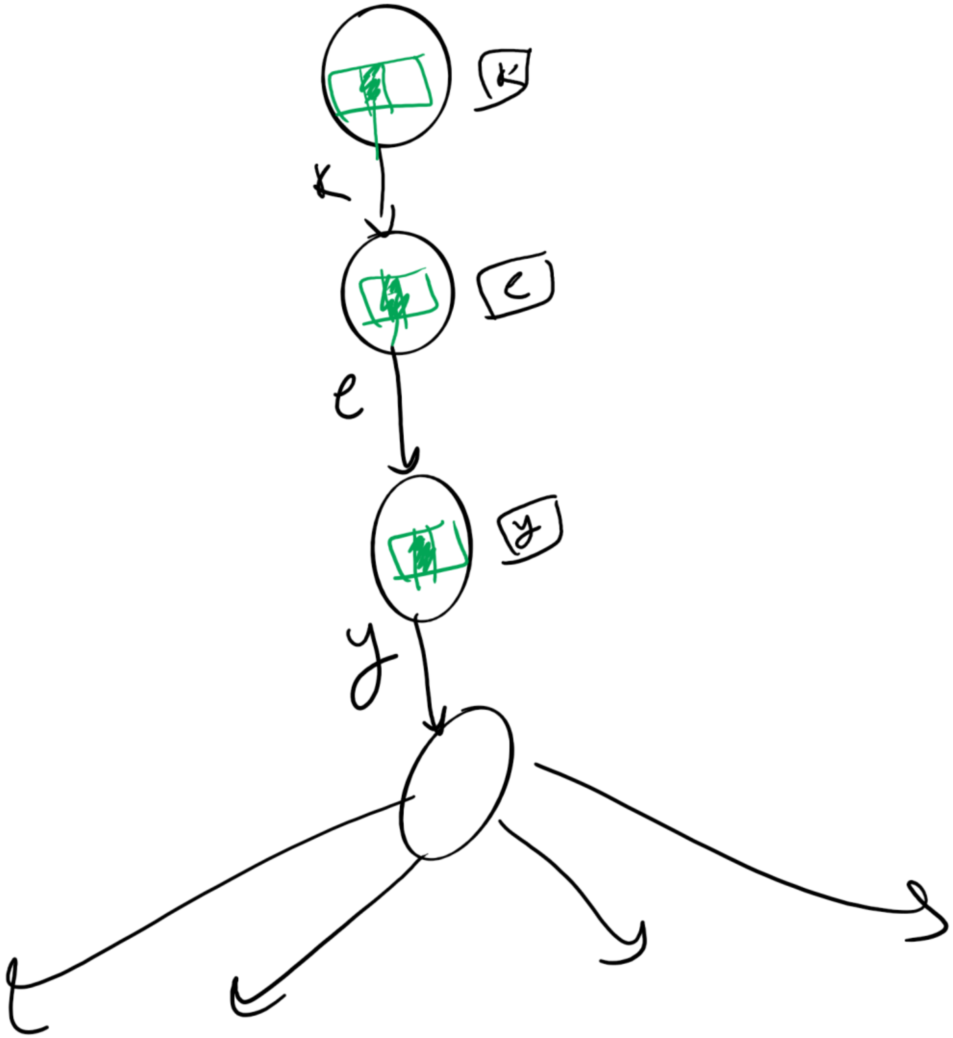


Summary of running time

	insert	search hit	search miss
binary RST	$\Theta(b)$	$\Theta(b)$	$\Theta(\log_2 n)$ on average
multi-way RST	$\Theta(w)$	$\Theta(w)$	$\Theta(\log_R n)$

R-way RST's nodes are large!

- Considering 8-bit ASCII, each node contains 2^8 references!
- This is especially problematic as in many cases, a lot of this space is wasted
  - Common paths or prefixes for example, e.g., if all keys begin with "key", thats 255*3 wasted references!
  - At the lower levels of the trie, most keys have probably been separated out and reference lists will be sparse



Solution: De La Briandais tries (DLBs)

Main idea: replace the array inside the node of the R-way trie with a linked-list

DLB Nodelets

Two alternative implementations:

```java
private class DLBNode {
    private Object val;
    private T character;
    private Node sibling;
    private Node child;
}
```

If search terminates on a node with non-null value, key is found; otherwise, not found.

```java
private class DLBNode {
    private Object val;
    private Character character;
    private Node sibling;
    private Node child;
}
```

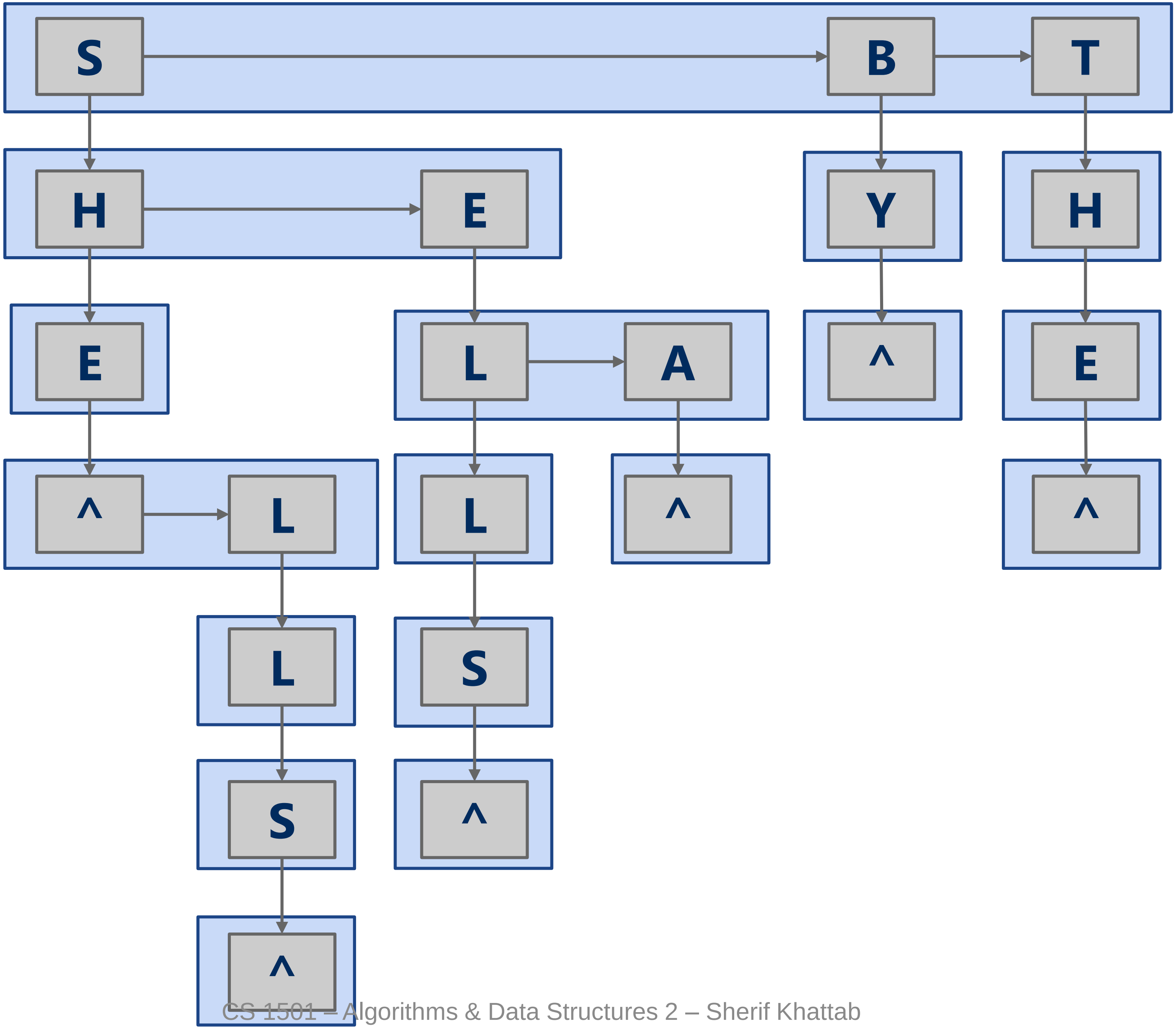
Add a sentinel character (e.g., ^) to each key before add and search

If search encounters null, key not found; otherwise, key is found

Adding to DLB Trie

- if root is null, set root ← new node
- current node ← root
- for each *character c* in the key
  - Search for *c* in the linked list headed at current using sibling links
    - if not found, create a new node and attach as a sibling to the linked list
  - move to child of the found node
    - either recursively or by current ← child
- if at last character of key, insert value into current node and return

DLB Example



DLB analysis

- How does DLB performance differ from R-way tries?
- Which should you use?

	search hit insert	
R-way RST	$\Theta(w)$	
DLB	$\Theta(wR)$	

Runtime Comparison for Search Trees/Tries

	Search hit	Search miss (average)	insert
BST	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
RB-BST	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
DST	$\Theta(b)$	$\Theta(\log n)$	$\Theta(b)$
RST	$\Theta(b)$	$\Theta(\log n)$	$\Theta(b)$
R-way RST	$\Theta(w)$	$\Theta(\log_R n)$	$\Theta(w)$
DLB	$\Theta(wR)$	$\Theta(\log_R n \cdot R)$	$\Theta(w \cdot R)$

Final notes on Search Tree/Tries

- We did not present an exhaustive look at search trees/tries, just the sampling that we're going to focus on
- Many variations on these techniques exist and perform quite well in different circumstances
  - Ternary search Tries
  - R-way tries without 1-way branching
- See the table at the end of Section 5.2 of the text